

AD-A044 915

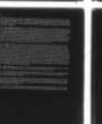
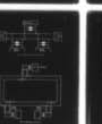
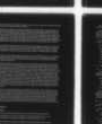
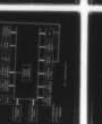
ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 3/1
A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CON--ETC(U)
MAY 77 G E SCHWEIZER, A A CALLAWAY, E C GANGL

UNCLASSIFIED

AGARD-AR-90

NL

1 OF
ADA
044 915



2

AGARD-AR-90

AGARD-AR-90

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD ADVISORY REPORT No. 90
**A Study of Standardization
Methods for Digital
Guidance and Control Systems**

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DDC
OCT 5 1981
C

NORTH ATLANTIC TREATY ORGANIZATION



DISTRIBUTION AND AVAILABILITY
ON BACK COVER

A STUDY OF STANDARDIZATION METHODS
FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS

ADA 044915

AD NO.
DDC FILE COPY

GCP

NORTH ATLANTIC TREATY ORGANIZATION
ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT
(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

12 555 P.

9
AGARD Advisory Report No. 90
6
A STUDY OF STANDARDIZATION METHODS
FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS.

10 G. E. / Schweitzer
A. A. / Callaway
E. C. / Gangl
B. Vandecasteele
J. N. / Bloom

11 May 77

JB

THE MISSION OF AGARD

The mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

- Exchanging of scientific and technical information;
- Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;
- Improving the co-operation among member nations in aerospace research and development;
- Providing scientific and technical advice and assistance to the North Atlantic Military Committee in the field of aerospace research and development;
- Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field;
- Providing assistance to member nations for the purpose of increasing their scientific and technical potential;
- Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Program and the Aerospace Applications Studies Program. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

Published May 1977

Copyright © AGARD 1977
All Rights Reserved

ISBN 92-835-1244-8



*Set and printed by Technical Editing and Reproduction Ltd
Harford House, 7-9 Charlotte St, London, W1P 1HD*

Report prepared by members of the Working Group:

Dr G.E.Schweizer (Chairman)	Germany
Dr A.A.Callaway	UK
Mr E.C.Gangl	USA
M. l'Ing. B.Vandecasteele	France
Mr J.N.Bloom	Canada
Dr W.Metzdorff	Germany

Report edited by: Dr A.A.Callaway.

SUMMARY

This Report contains the findings of the AGARD GCP Working Group No.02, set up to investigate standardization methods for digital guidance and control systems, particularly with regard to data transmission techniques and high level programming languages. It includes discussion of the general problems and techniques, reports on the particular experiences of the individual nations, and concludes that, whilst much work remains to be done on software aspects, the field of data transmission may be amenable to early standardization.

The Annexes to the Report contain full details of the techniques studied, and include comparisons of data transmission methods and high level languages. These comparisons are not intended as quantitative assessments but are designed to outline the relevant features of the different techniques.

ACCESSION for

NTIS ☐ Vile Section ☐

DDC ☐ B.II Section ☐

UNANNOUNCED

CLASSIFICATION

BY

DISTRIBUTION/AVAILABILITY NOTES

614

A

CONTENTS

	Page
SUMMARY	iii
	Reference
1. INTRODUCTION	1
2. BASIC SYSTEM ASPECTS AND PROBLEMS	3
2.1 The Advantages of Digital Techniques	3
2.2 The Need for Digital Standardization	4
2.2.1 General	4
2.2.2 Hardware Standardization	4
2.2.3 Software Standardization	8
2.2.3.1 The current problem	9
2.2.3.2 General criteria	9
2.2.3.3 System description language	9
2.2.3.4 Real-time features (tasking)	9
2.2.3.5 System programming capability (operating system)	10
2.2.3.6 Summary	10
3. US EXPERIENCE	11
3.1 Introduction	11
3.2 A Justification for the Need of Standardization in Digital Avionics	11
3.3 Multiplex Data Bus Experience	13
3.4 Higher Order Language Experience	13
4. FRENCH EXPERIENCE	
4.1 Multiplex Data Busses	16
4.1.1 Provisional Recommendations for a System of Multiplexed Transmission along Bus Lines in Aircraft	16
4.1.2 Digital Data Busses for Combat Aircraft	17
4.1.2.1 General	17
4.1.2.2 Applications	17
4.1.3 Advantages of Digital Data Busses	17
4.2 High Level Languages	18
4.2.1 LTR	18
4.2.2 Applications	18
4.2.3 Advantages of High Level Languages	18
5. GERMAN EXPERIENCE	19
5.1 The German Experimental Guidance and Control Programme for Helicopters (HSF)	19
5.1.1 Background	19
5.1.2 The Problem	19
5.1.3 Mission Requirements for Demonstration of the System	19
5.1.4 System Philosophy	20
5.1.5 System Architecture	22
5.1.5.1 Sensors	22
5.1.5.2 Control and display	22
5.1.5.3 Signal processing	22
5.2 Future Software Considerations	24
6. BRITISH EXPERIENCE	24
6.1 Data Transmission	24
6.2 High Level Language Standardization	25
7. CONCLUSIONS AND RECOMMENDATIONS	26
7.1 Concluding Remarks	26
7.2 Recommendations	27
Annex A THE MULTIPLEX DATA BUS	A-1
Annex B MIL-STD-1553A: AIRCRAFT INTERNAL TIME DIVISION COMMAND/RESPONSE MULTIPLEX DATA BUS (USA)	B-1

	Reference
Annex C PROVISIONAL RECOMMENDATIONS FOR A SYSTEM OF MULTIPLEXED TRANSMISSION ALONG BUS LINES IN AIRCRAFT (FRANCE)	C-1
Annex D DEFINITION OF THE DIGIBUS FOR THE EXCHANGE OF DIGITAL DATA IN A COMBAT AIRCRAFT (FRANCE)	D-1
Annex E COMMON STRUCTURE AND INTERFACES FOR GUIDANCE AND CONTROL SYSTEMS WITH DIGITAL DATA PROCESSING (GERMANY)	E-1
Annex F BUS ASSESSMENT AND COMPARISON	F-1
Annex G JOVIAL J.73/1 (USA)	G-1
Annex H JOVIAL B-1 AVIONICS SUPPORT SOFTWARE (USA)	H-1
Annex I LTR MAXIRIS REAL-TIME LANGUAGE (FRANCE)	I-1
Annex J PEARL SUBSET FOR AVIONIC APPLICATION (GERMANY)	J-1
Annex K EXTRACTS FROM THE OFFICIAL DEFINITION OF CORAL 66 (UK)	K-1
Annex L CORAL 66 FOR SOFTWARE AND REAL-TIME APPLICATIONS ON SMALLER COMPUTERS (UK)	L-1
Annex M A COMPARISON OF THE PROGRAMMING LANGUAGES: CORAL 66, JOVIAL, LTR AND PEARL	M-1

A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS

1. INTRODUCTION

In recent years, the AGARD Guidance and Control Panel (GCP) has organized a number of symposia to review the state of the art, the problems and the shortcomings of present-day guidance and control systems.

The symposia have provided evidence of a massive spurt forward in the operational requirements for guidance and control systems which is reflected in the development of new system philosophies. Examples are: all-weather, area navigation, blind landing, air traffic control, remote piloting, weapon delivery, better use of high performance aircraft (CCV) and better flying qualities over a wide flight envelope. Emphasis is also placed more and more on the fact that these new systems must be operable by smaller crews.

These advancing requirements have given rise to the need for new analytical methods in the investigation of problems, and modern concepts of automatic control science have been successfully applied to solving the problem. Examples here include optimized control, modern digital filtering methods, matrix methods, adaptive control, time domain methods, and so on.

Joint symposia with the Flight Mechanics Panel have demonstrated that flight mechanics and guidance and control specialists are currently using the same basic analytical engineering methods to formulate the problems in mathematical terms. In the past, the methods employed by a mechanics specialist investigating flying qualities or aircraft motion were quite different from those used by a control engineer working on a flight control system. There are no longer major difficulties in interfacing flight mechanics problems with guidance and control problems. This was seen, for example, during recent symposia on Adaptive Control and on the Impact of Active Control Technology on Airplane Design.

The GCP symposia have also revealed that new operational requirements are dictating the use of sophisticated digital systems at the same time as these become feasible propositions, due to advances in solid state technology (MSI, LSI, etc.). Today, digital computation and data processing are the fundamental keys to many guidance and control systems. Likewise, digital techniques are applied to sensors, transducers and interfaces.

Papers and discussions at the symposia, however, have exposed severe shortcomings in the use of digital systems. It has been found that many digital avionic systems have been developed unilaterally, with little overall system design philosophy. The fact that each major subsystem is often functionally autonomous, or even isolated, from the other subsystems can cause serious problems, since no part of any subsystem can be shared with another. Figure 1.1 depicts the state of the art in many of today's operational systems, where there is abundant hardware redundancy of processors, control and display units, with no sharing of system components and, consequently, no functional redundancy.

Reductions, by an order of magnitude, in component cost and size have enabled designers to increase the complexity of systems and, above all, subsystems. As a result, overall system reliability has decreased in spite of increases in component reliability.

Some of the symposia showed that several very ambitious and complex digital guidance and control systems have resulted in disappointment. As complete real-time systems they became operational only after various delays, and in some cases they even had to be dismantled and removed. Other experiences in the design of complex computer-based systems have shown that development time, and cost of procurement and operation, proved to be much higher than was anticipated at the time of their conception.

These difficulties and near-failures are usually counted as the price of pioneering; however, the study of these problems reveals that their source does not necessarily lie in the technology, but in the lack of system thinking and system engineering methods for real-time computer-based systems.

It is not that the elemental subsystems of digital real-time guidance and control systems are any more complex than other parts, like sensors or conventional analog equipment, nor are these problems caused by the on-board computer hardware — most of the hardware can be regarded even as "off the shelf". The problem arises because of the lack of any standards for defining and interconnecting the subsystems of digital real-time guidance and control systems. The difficulties of interfacing and integrating digital subsystems into a complex system have, in general, been severely underestimated, due mainly to the assumption that the individualistic problem-solving approach, appropriate to the design of a particular subsystem, is equally applicable to building the complete system, leading to the above-mentioned lack of digital system philosophy.

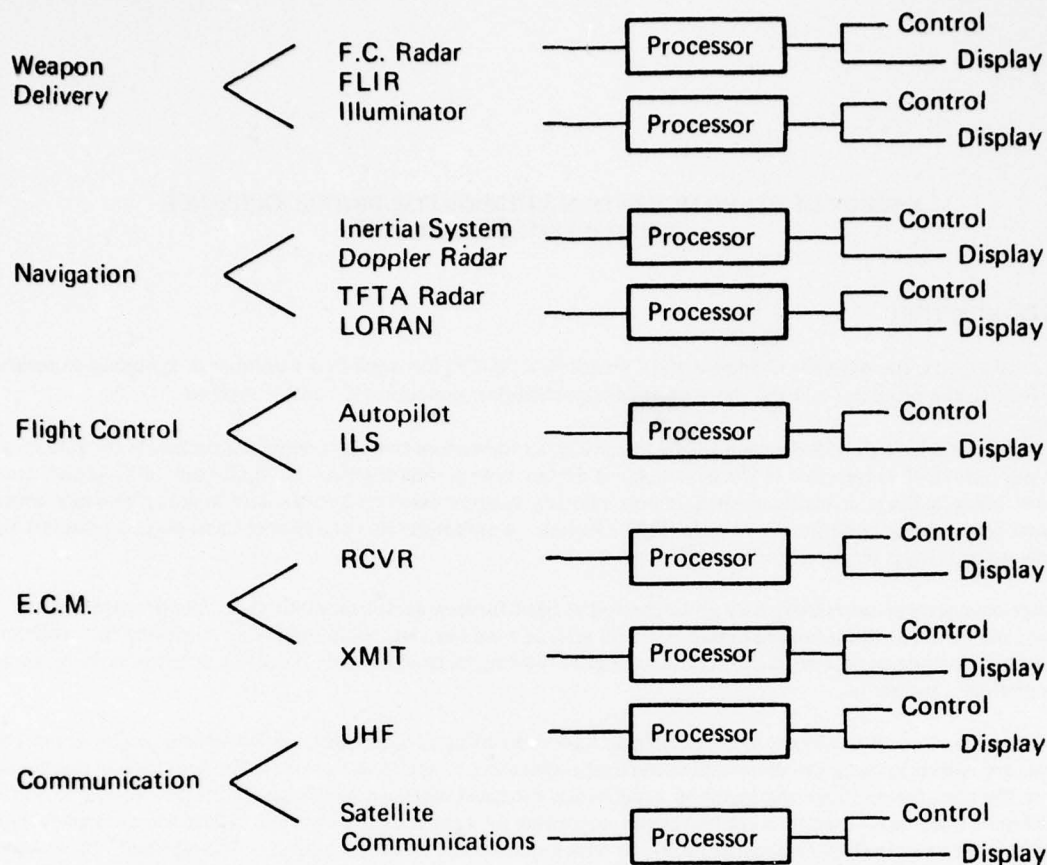


Fig.1.1 System architecture of today's guidance and control and avionics system

From the history of the development of major digital guidance and control systems, one recognises that an important factor in the complexity of real-time computer-based systems is the software involved. The symposia and discussions have shown that there is, as yet, no organized and agreed body of knowledge concerning the design and coding of real-time computer programs. The lack of engineering discipline and management foresight has resulted in the present situation, where individual programming skill dominates over team-work.

In the development of large software systems by team-work, where software subsystems must be integrated, it is essential that programs be readable and understandable. The present lack of any standard method for defining individual software modules and interconnecting them into a large software package, however, causes serious problems. Other problems are caused by the lack of communication between programmers and system designers. The designer of an operational avionic system has, as a rule, little experience with real-time programming, and the typical computer programmer knows little about guidance and control problems of aircraft, space vehicles, missiles or drones. The dearth of methods for the specification of software systems within the guidance and control community, as well as the avionics world in general, creates a situation of minimal understanding of the designer's real intentions and requirements by the programmers who must implement the software.

Discussions during several of the business meetings of the GCP indicated that, in an attempt to overcome some of these problems, several NATO countries were investigating the introduction of standard interfaces for digital hardware and software modules. The idea behind these standardization efforts is to aim for some commonality in the architecture of on-board systems and for compatibility among on-board as well as external ground subsystems or systems.

The Panel was of the opinion that, within the guidance and control community, it was highly desirable to analyse the various proposed standards and to disseminate the information. This information would enable specialists to evaluate the usefulness of existing or proposed digital airborne standards with respect to their applicability to guidance and control systems. Furthermore, it was thought that this dissemination of proposed standards could help to avoid the unnecessary proliferation of too many standards, which could lead to confusion and certainly not to eventual architectural commonality and system compatibility.

The Guidance and Control Panel, then, decided to set up a Working Group to study the efforts for standardization of digital guidance and control systems. The Working Group was established in 1974, and its terms of reference were:

- (1) To examine the various proposed or existing standards defining the requirements for data transmission techniques and real-time computer languages utilised in aircraft weapon systems.
- (2) To consider them in the light of present and probable future requirements for guidance and control.
- (3) To develop plans for disseminating the research and development plans of various countries dealing with standardization of digital aircraft weapon systems.
- (4) To establish tools and methodologies for evaluating the usefulness of existing or proposed digital airborne standards, with respect to their applicability to guidance and control systems.
- (5) To prepare recommendations for AGARD Headquarters concerning standards related to guidance and control, and to indicate the conclusions to be drawn from these recommendations.

This Report, then, presents the deliberations and findings of the Working Group. Section 2 considers the potential advantages of the use of digital techniques and discusses how some of these may not be realised without the concept of standardization. This is then expanded in terms of hardware aspects, particularly with regard to interfaces and data transmission, and software aspects of high level language standardization.

Following this general discussion, there are four Sections which consider experiences within individual NATO nations: Section 3 examines US experience; Section 4, French experience; Section 5, German experience, and Section 6, British experience. The main text of the Report is then rounded off with the conclusions and recommendations of the Working Group, presented in Section 7. This Section expresses the view that software problems – in common with other problems involving digital computation – require more study in a forum which is not confined simply to guidance and control systems, but the field of data transmission, being more advanced in terms of individual standardization attempts, may be amenable to early NATO standardization attempts.

The comprehensive set of Annexes comprises the information studied by the Working Group in drawing up its recommendations. Annex A is a general description of multiplex data bus methods, and Annexes B to E give details of such data standards as have been formulated within several NATO nations. Annex F is a comparison of these data bus implementations, intended not as a quantitative assessment but as an outline of their relevant features. Annexes G to L give details of the high level programming languages in use for real-time applications in the different countries, and Annex M is another non-quantitative assessment, this time in the form of a questionnaire, comparing the features of the languages.

2. BASIC SYSTEM ASPECTS AND PROBLEMS

2.1 The Advantages of Digital Techniques

Operational needs for guidance and control equipment, and a dramatic change in technology, induced by the rapid development of integrated circuits, have led to the adoption of digital systems in the field of guidance and control.

Some of the potential advantages of digital techniques in guidance and control systems for operational use are:

- precision and range of computing power (including sophisticated algorithms);
- techniques for filtering (e.g. Kalman filtering for navigation);
- stability of the encoded data (i.e. no drift);
- possibility of storing information (libraries for navigation, aircraft, engine data, etc.);
- improved man-machine interfaces (keyboards, electronic displays, standard peripherals, etc.);
- wider range and automation of mode selection;
- possibilities of built-in test equipment and fault detection;
- range of mechanising methods for fault-tolerance;
- multiplexing of data channels (reduced weight and increased flexibility).

In many cases these operational advantages make the use of digital computer-based systems essential to meet today's requirements for guidance and control.

From the viewpoint of system architecture, digital techniques offer a number of additional benefits, some of these being:

- the implementation of hierarchical systems;
- the building-block approach to system architecture;
- compatible hardware resulting from standard interfaces;
- compatible software due to standard practices;
- modular reusable hardware;

- system integration, mainly through software;
- changes in system operation by software;
- ease of upward and downward modification;
- replacement of modules, to take advantage of new technologies.

Developments in MOS semiconductor technology, thick film and thin film technology, printed circuit connection techniques, circuit encapsulation techniques and the availability of electro-optical techniques, all of which are less sensitive to the physical environment than former technologies, have considerably widened the range of digital systems applicable to guidance and control problems.

2.2 The Need for Digital Standardization

2.2.1 General

Avionic hardware is normally packaged in the form of equipment modules (so-called black boxes) which are interconnected via plugs and sockets and a wiring harness. The modularity is dictated by the function a module has to perform and by the requirements of maintenance. If a box fails it must be capable of being easily and economically replaced by an identical line replaceable unit (LRU).

This general approach has enabled the Air Forces and the Airlines to specify functions, mechanical dimensions, plugs and sockets, and allow the manufacturer reasonable freedom in choosing the technology and in design ingenuity for the boxes. It is now common practice to fit particular equipment, e.g., radio communication or Tacan, into a variety of aircraft types, thus reducing the necessary amount of training and logistic support.

The particular equipment chosen, however, such as an inertial attitude platform and a Doppler radar for navigation, cannot, in general, be replaced by different equipment, such as a complete self-contained inertial navigation platform, without excessive on-board re-wiring. That is to say, new equipment cannot, in general, easily replace equipment in established systems and subsystems. Furthermore, established systems cannot easily be equipped, during their life time, with additional sensor and processor modules to meet new operational goals.

As a result, the specifications for the data transmission between the units have often been established on an individual application basis. This has led to a proliferation of data transmission techniques within the individual countries, and within NATO, which invalidates all of the system advantages listed above. The lack of interface standards linking different equipment into an integrated system, which meets specific requirements, leads to individualistic system approaches without modular reusable hardware.

The situation in the area of software is even worse. It is fair to say that there is, as yet, no organized and agreed body of knowledge concerning program design for real-time computer-based guidance and control systems. This means that the system engineer (usually not a software expert) has no strict rules available to specify the software for a given problem. As a result, the programmer has to study and understand the actual control problem before being able to lay down the software design. Thus, the individual programmer is more or less allowed to dominate and determine the design of the software philosophies, and by individually tailoring all programs he automatically prevents the generation of modular reusable software.

2.2.2 Hardware Standardization

A digital real-time computer-based on-board system should not be designed by adding equipment piecemeal, each item performing the special function for which it is designed. This approach leads to unnecessarily complex systems, resulting in high cost and many of the other problems mentioned in Section 1. Instead, digital systems for guidance and control should be architected in a similar manner to real-time process control systems, usually in a hierarchical form. A typical configuration is shown in Figure 2.1.

If full use is to be made of the potential of digital computer technologies, the completely separate implementation of the different functions of an avionic system as demonstrated in Figure 1.1 is not the appropriate solution. A function, in this context, means a subsystem, such as a weapon delivery system or part of such a system. The approach for the design of an avionic system must be to implement the subsystems functionally within one integrated system, where processors, the central computer, if such exists, the sensors, and the presentation and control part may be shared. Redundant sensors, processors and other hardware should be provided only if dictated by the overall operational reliability requirements, and not by the hardware implementation of individual functions as in Figure 1.1.

Functionally, digital subsystems usually comprise a sensor part which provides input data, a data conversion part which converts the input data into the digital formats required, a data acquisition and processing part, and a presentation and control part, which provides the crew with the necessary information to take actions for proper operation and for monitoring, fault detection and testing.

If the different functions (flight control, navigation, etc.) are designed to be performed within one computer-based digital avionics system and not necessarily by dedicated equipment, data routing becomes a primary pre-requisite for

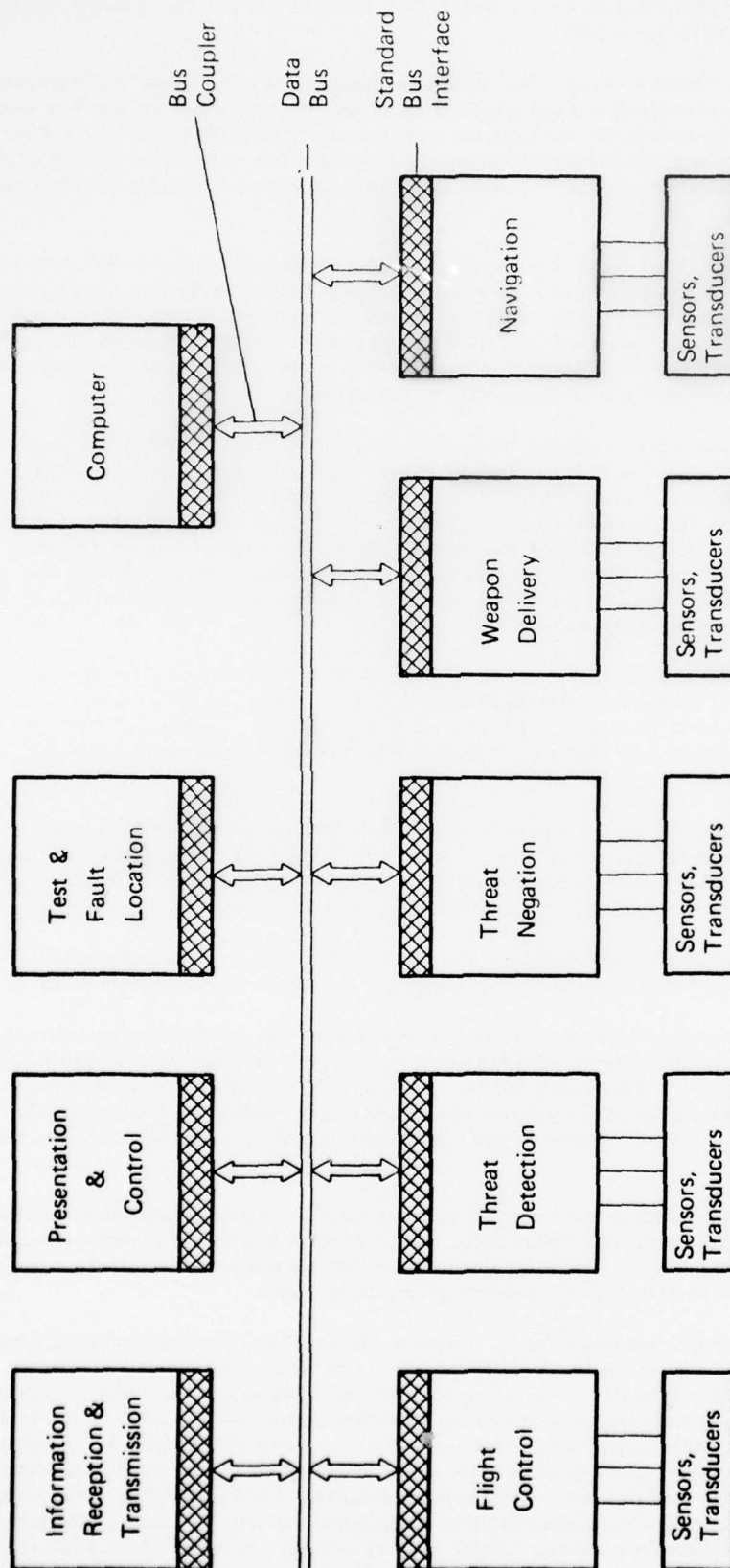


Fig.2.1 System architecture of future computer based guidance and control and avionics on-board system

proper performance, as indicated in Figure 2.1. Such a systems approach allows sensors to be shared functionally by several subsystems. For example, the information from the inertial sensors may be used by the navigation, flight control and weapon delivery subsystems. The processing function may be undertaken by a centralised computer, or the centralised computer can be substituted by a federated network of decentralised processors, at least for emergency operation in case of failures. The same applies to the memories.

Figure 2.1 indicates that a good deal of the design effort for future digital guidance and control hardware must be devoted to the definition of operational requirements and system design parameters, such as information flow, information rate and partitioning of processing loads on the one hand, and interface requirements for data routing and communication with the analogue world, via sensors and transducers, on the other. In the future there will be a strong interdependence between the subsystems and functions of a digital real-time avionic system, with various needs for data acquisition and data distribution.

The so-called "star" configuration (Fig. 2.2), which has been applied, for example, to the European Tornado (MRCA) aircraft, used to be the preferred solution to the general problem of data acquisition and distribution. In this configuration, each data source and user is connected to a central controller — usually the central computer — by its own dedicated line. The integrity of this central authority, then, is essential for the survival of the entire system and must be maintained at all cost. Network complexity is high, and useful application is restricted to a distance of a few metres if excessive wiring is to be avoided. System expansion or modification usually requires rewiring.

The principle of "bus" configuration (data highway), representing a modern approach to system interconnection, is depicted in Figure 2.3, and is described in detail in Annex A. Transmission of data along the bus can be performed in several ways; however, digital time division multiplexing (TDM) is the most common method for aerospace applications. The bus configuration is well suited to the case of the distributed system, where system components are spread over a relatively wide area, and the possibility of incorporating system control also in a decentralised manner provides a "fail-soft" feature which is of vital importance in military applications. In addition, wiring is kept to a minimum, and system expansions and retrofits do not normally require rewiring or bus controller modifications, provided that address decoding is decentralised and distributed to each source and user.

In the case of distributed systems involving long distances such as may be required within an aircraft (up to 100 m), only serial bus operation is practically feasible to avoid the problems of excessive wiring and the resulting weight penalties. With present-day technology, data rates of the order of 1 Mbit/sec can be achieved on a serial bus, and this is generally sufficient to satisfy most avionics requirements. High frequency signals, such as video signals which may be fed through coaxial lines, have not been considered here.

The bus structure shown in Figures 2.1 and 2.3 is particularly suited to the use of equipment from different suppliers, and the addition of new modules at any given time. To meet these goals, however, there must be an agreed body of specifications for bus communication procedures as well as for the electrical and mechanical design of interfaces. In other words, standards for multiplex data busses must be set up to achieve these goals.

Recognizing this, several NATO countries have made proposals for standards for a serial multiplex data bus. These proposed standards are given in Annexes B to E, and a technical assessment and comparison is shown in Annex F.

A serial data bus is an effective means of interfacing digital guidance and control modules on an equipment level using modern technologies. As indicated above, proposed data rates of the order of 1 Mbit/sec seem to be sufficient for the requirements of guidance and control systems as long as some special services, such as TV communications, are excluded. Similar systems approaches are discussed within the commercial airlines' engineering divisions and the ARINC committees. The efforts there have not, so far, resulted in firmly proposed standards for universal data busses, but they should be observed closely.

If guidance and control subsystems are to be further partitioned into suitable sub-modules, standard electrical interfaces and data transmission must be provided between them. Sub-modules can be analogue to digital converters, registers, memories, microprocessors, and so on. This further partitioning from the equipment level to the module level may be necessary if modules are to be supplier-independent line replaceable units.

Functional partitioning of equipment has severe impacts; therefore, trade-off areas involved in selecting levels of functional partitioning for avionic equipment must be carefully studied. Trade-off areas are system performance, reliability, module size, system space, weight, maintenance concepts, physical packaging, growth potential, inter-system commonality and others. There is a lack of experience concerning the trade-off areas; however, there is strong evidence within some NATO countries that a standard electronic module programme may be beneficial to the Air Forces, and the primary economic benefits of standardization would be in the area of support costs. These potential savings are realizable only if there is a large amount of replication of the standard functions. The digital and signal processing functions appear to be special candidates for standardization, with the provision that these functions can perform a variety of signal and data processing operations. Microprocessors, memory modules and register blocks belong to this category. For all kinds of data conversion functions, such as analogue to digital, digital to analogue, synchro to digital converters, multiplexers, and the like, a large amount of replication can be expected also. These functions play a major role within guidance and control systems.

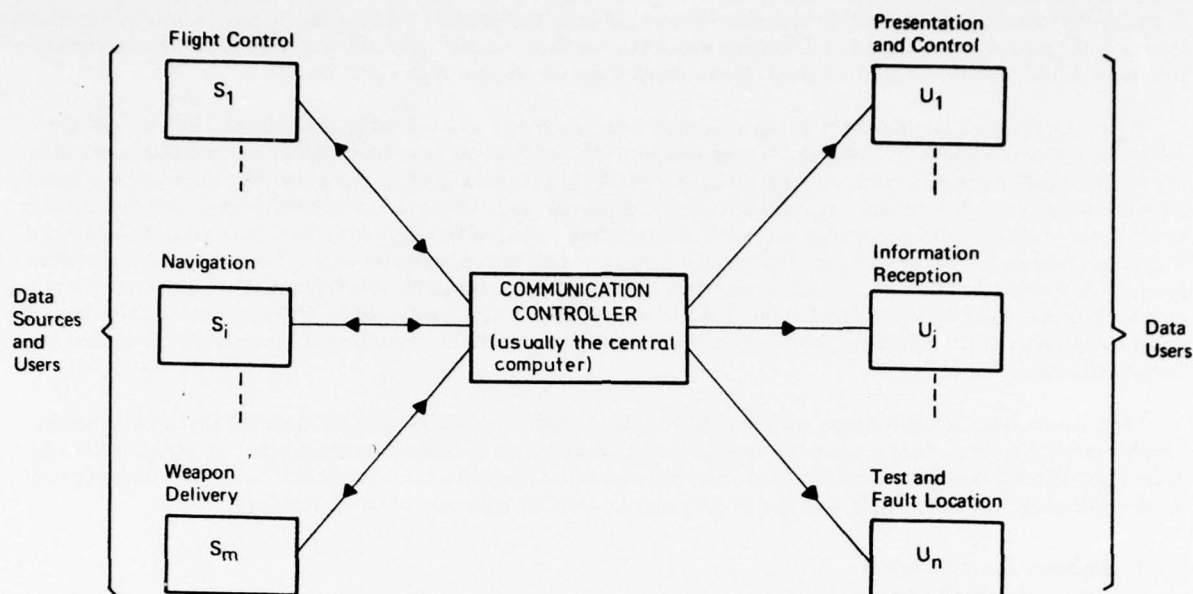


Fig.2.2 Star configuration in a guidance and control system

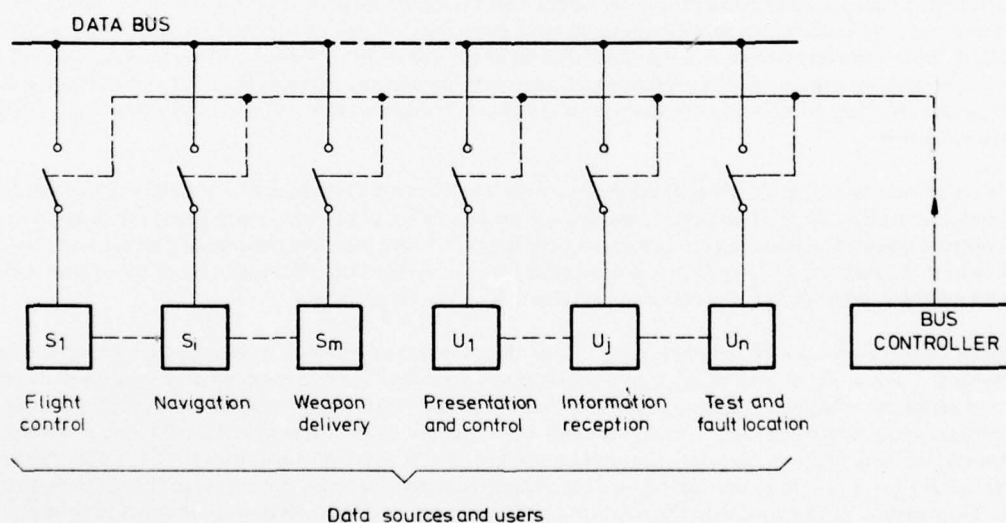


Fig.2.3 Bus (data highway) configuration

In addition to the specifications of the functions within a standard electronic module programme, mechanical dimensions, interconnection technology, and standards for interfacing and data transmission must also be provided if functional partitioning of guidance and control systems is to be a standard aim. If digital signal processing functions and data conversion functions were to be included in a standard module programme there would be no problem in finding functions that would, indeed, result in high inter-system and intra-system commonality, and, therefore, lead to significant savings in the support area. Outside the avionics world this has been demonstrated universally by the nuclear community with the CAMAC system, a digital standard module programme for laboratory applications.

If the Air Forces retain the ATR box as a standard, the circuit card area will be of the order of 150 cm², and the connector will have somewhere between 100 and 150 pins. The cards of modules (one module can comprise more than one card) should be equipped with standard interfaces which must provide fast data transmission, and, in order to interface modules within a box or rack, standard bus structures are required. With present technology this must be a parallel bus in order to achieve sufficiently high data rates, although the advent of fibre optics for data transmission could make a serial bus feasible for this application. One standard for a parallel bus has been proposed in Germany, and is described in Annex E, Section 3. The USA, in conjunction with the Standard Electronic Module (SEM) programme, is currently investigating the use of a parallel data bus internal to the LRU and a definition should be available shortly. Also, the UK, under the auspices of the Admiralty Surface Weapons Establishment (ASWE), is pursuing its own parallel peripheral interface programme.

The inverse relationship between space/weight and standardization appears to be the trade-off that most concerns people. It is certain that there is a penalty in space and weight inherent in module standardization practices. Although there is insufficient experience available so far, one can conclude, from some pilot demonstrations with attitude control systems for spacecraft and aircraft, that this penalty may be bearable with present-day technology.

2.2.3 Software Standardization

2.2.3.1 The current problem

Real-time computer-based guidance and control systems depend equally on software as on hardware. In reality, the main rationale for implementing a system with software instead of dedicated hardware is flexibility. Software is used when identical hardware (such as a processor) is to be used for a variety of required functions, or if the required function may be replaced by another function requiring a different program, or if the required function is so complex that many changes or modifications can be expected before an operational state is reached. The software approach is, furthermore, the proper solution when the system is expected to grow and to be revised or modified continuously, or when the system is sufficiently unique that the economics are in favour of the software approach instead of developing special hardware.

Basic software commonality in scientific and commercial data processing applications is state of the art today, although there was a time when commonality in these areas seemed as unlikely as it now does in real-time digital computer-based applications in weapon systems, in avionics, and in guidance and control. Basic software packages are not available for all the requirements of these real-time systems and, as a consequence, the inherent flexibility of software is seldom realised in practice in these areas. Only too often, any changes or modification to software leads to a change in the entire software design.

There are at least two basic differences between a conventional stand-alone scientific system and a computer-based on-board real-time system. In the latter, the computers or processors are physically incorporated into a larger system (sensors, controls, displays, actuators, etc.) whose primary function is not just data processing, furthermore, the sequence of the operations the software has to perform is determined largely by the state of the process of the system and cannot be rigidly scheduled in advance as is the case in most scientific computer programs.

Software commonality can be achieved only through the adoption of common programming languages, since software tools and aids are built around selected programming languages. There are special problems involved with the selection and adoption of high level languages for universal use in computer-based guidance and control systems. A common programming language must be useful for many diverse applications in the avionics field and, probably, also in the weapon system field, and this leads to a large-volume language with many possible subsets. The result can be a language expensive for everyone to implement, understandable to none and, thus, not suited to the application of real problems. The question is how to satisfy the needs for both simplicity and specialisation to avionics problems, simultaneously in the same programming language.

Today, the most pressing software problem is high cost — software cost is currently estimated at \$3.0 billion in the support and operation of software systems in the US Department of Defense. This figure must be compared with an estimated \$1.5 billion for computer hardware purchase in order to appreciate the importance of the problem. This high cost is mainly the result of the lack of transparency, the dearth of software techniques for enabling the software implementer to understand the designer's real requirements, the lack of well suited real-time high level languages which do not force the programmer to use the dangerous capabilities of self-modifying machine programs, and machine-dependence of the programs with the additional need for individually tailored drivers to any process peripheral (such as converters, counters, external memories and processors).

Software commonality for on-board computer-based systems is urgently needed. There is presently no internationally accepted language for these applications, although the UK has pursued a defence standard language policy for some 10 years, employing the language CORAL 66. This language has been widely used for naval and land-based systems, but has only recently been applied to avionic systems. One of the criticisms of CORAL 66 is that, because it is designed to interface with machine-specific operating systems, it has no specific inherent real-time features, and it is to be expected that a large number of users in the NATO nations would adopt a new real-time language for on-board avionic systems if other language design criteria were selected.

Quite often, criteria in the software field tend to be general and vague and are, therefore, not fit for quantitative evaluation. In spite of this fact, it is believed that the few criteria, mainly concerning software problems for real-time applications, discussed below should give some insight into the problems, the present state of the art, and some future trends, to the guidance and control community.

2.2.3.2 General criteria

A common real-time programming language to be used on a large number of different computers embedded in various avionic on-board computer-based systems must be machine-independent if there is to be any chance for standardization and program portability. On the other hand, programs written for machines having different memory sizes, different peripheral configurations, specialised hardware capabilities, etc., obviously cannot be completely hardware independent. Additional difficulties in avionic systems are caused by the fact that they are not like management data retrieval systems; which are dominated by big general purpose processors and so-called standard peripherals like magnetic tapes, discs, punch-card equipment, etc. Guidance and control computer-based systems are integrated systems with various computers and data processors for different sizes, a variety of process peripherals for data conversion and data acquisition, access to the analogue world via sensors and transducers, and manifold data routing between all the system components.

The program sequence is very much influenced by external events. The main characteristics of a digital guidance and control or avionic real-time on-board system (as compared to a conventional automatic data processing system) are that the computers and processors are incorporated into larger systems whose primary functions are not necessarily data processing, in the conventional sense, but data distribution and acquisition, data presentation and display, test and fault location, and others. This category of real-time system, to which digital guidance and control systems undoubtedly belong, is known as the *embedded computer system (ECS)* in the literature.

What is needed in the case of digital avionic systems of various configurations, is the ability to write programs which are somewhat machine- and strongly system-dependent (sensors, data conversion peripherals, etc.) in a machine- and configuration-independent manner at a real-time language level. Provisions must be made so that the program written on the machine- and configuration-independent language level can have access to the individual computer, data processors and all other components (sensors, actuators, etc.) of an on-board digital system, via code generators and the utilisation of the system generation capabilities (e.g. modular operating system features).

2.2.3.3 System description language

Software for computer-based on-board avionic systems is often unresponsive to the designer's and the user's needs. A first requirement for better understanding of the intended operation of a digital avionic system is a full description of the configuration of the whole system. The present-day method consists of natural language (English, German, French, etc.) accompanied by block diagrams and flow charts. The requirement here is for a set of syntax and semantics, comparable to a computer language, for the description of the total hardware configuration of the digital avionic system, with symbolic names for each element (e.g. sensors, converters, processors, etc.) to be called up under program control. This is what is called a *system description language*, and it has several advantages:

- The given configuration and hardware operation of the avionic system is documented in the computer program itself in a readable form and, thus, provides for better responsiveness.
- Avionic system configurations and required system operations are manifold. With the aid of the system description language, the generation of an operating system for the particular application is possible.
- The system description language is an appropriate interface between the system designer and the software engineer. The language is the tool the design engineer uses to describe the static state of the designed system configuration, data paths, direction, and so on, in an agreed and unambiguous manner.

2.2.3.4 Real-time features (tasking)

The ability to control the real-time behaviour of a system is essential to all applications within guidance and control. Avionic systems generally consist of a number of program tasks which run autonomously or are influenced by external events. Software elements are required for the supervision and control of such tasks. When a high level language is selected as a candidate for a standard for guidance and control applications it would be desirable for it to have facilities for activating and scheduling parallel tasks, depending on spontaneous events. If the language is designed in this way there should be little or no need for its user to enter the machine language level.

By means of the real-time language elements, the design engineer should be able to describe the dynamic behaviour of the program, using a well-defined set of syntactic and semantic rules. It is to be expected that the design engineer is the man who specifies, and best knows, the external process (i.e. the guidance and control problem) and, therefore, the selected real-time language should enable him to specify the dynamic state of the system on the language level. A language whose tasking facilities can form the interface between the design and software engineers is, therefore, desirable.

If machine language insertions cannot be avoided, they should be encapsulated so that they can be readily recognised when moving to another object machine or when modifying the programs.

2.2.3.5 *System programming capability (operating system)*

Usually, the software of a general purpose computer system is divided into two sections: the application programs, and the means by which these programs are scheduled and computer resources are allocated. The latter is normally called the operating system. Operating systems for general purpose computers comprise all the required features for scheduling and for access to the standard peripherals. For real-time applications, additional program sections are required within the operating system to provide software access to real-time peripherals specific to the system, such as sensors, displays, converters, and so on.

Operating systems for conventional data processing systems are generally very large programs. With the addition of the required real-time features they, naturally, have the tendency to grow even larger. Many applications in guidance and control as well as in avionics use computers and data processors which cannot afford this overhead, and do not require the overall flexibility of a general purpose operating system. These applications require specialized modular executive systems.

There is, however, a severe lack of system engineering methods for better conceptual understanding of the problems, and for defining the operating system functions for real-time systems. If one employs the normal, present-day, methods when implementing special procedures for the required real-time functions, one can face the following difficulties:

- poor system organisation and, thus, low documentation value;
- no re-usable software modules;
- problematic software integration;
- difficulties in modifying software in response to changes in hardware configuration.

A different solution for the provision of a modular operating system could be achieved by embedding the operations for the real-time system into the language itself. During translation into object machine code, the required routines for the real-time system would be generated. This method automatically leads to fully portable systems; the complexity of the compiler, however, is considerably increased. Standardization of the hardware interfaces and the operation of the data routing on serial or parallel busses is undoubtedly of great assistance in the generation of portable systems.

2.2.3.6 *Summary*

The problems and criteria mentioned so far are specific to real-time systems. Other more general criteria, however, are important in the real-time area as well as in other areas. One of these is the requirement for a simple source language, since complexity in the source language can be a major cause of problems. In the development of software systems integrated from many separately developed parts, which must be capable of modification throughout their lifetime, it is essential for the programs to be readable and understandable by authors, integrators and maintainers.

In guidance and control as well as in avionics applications there are constraints to the size of the computer hardware; therefore, efficient code is necessary.

To sum up, then, the inadequacies of present software implementation methods for avionic systems have been stressed in this Section. It has been shown that one of the main problem sources is the lack of standardized high level programming language specifically suited to real-time process control applications, which is acceptable to all NATO nations.

It is apparent that these problems cannot be solved merely by defining another high level language. The reason is that conventional programming languages are designed for application on general purpose computers with standard operating systems and standard peripherals such as discs, tapes, printers, and so on. As opposed to general purpose machines, avionic systems represent sophisticated real-time systems involving a large number of application-dependent real-time peripherals; in addition, the flow of control is continuously altered by external events.

Such embedded computer systems set new standards for a high level language, the most significant of which are as follows.

- (i) A precise syntax must be provided for the description of the hardware configuration of actual avionic systems, including all real-time peripherals as well as the sensors and actuators.
- (ii) A standardized form must be defined for the description of external non-predictable events (such as interrupts) which influence the flow of control within a system of program modules.
- (iii) Tools must be established for the automatic application-dependent generation of operating systems for embedded computer systems in the avionics and guidance and control field. These operating systems should be generated on host machines of medium size, the main goal being extreme efficiency of storage space and running time on the target machine.
- (iv) In order to simplify system integration and debugging of future avionic software systems, modular programming techniques must be supported by the language.

If these demands are met, interfaces for the communication between system engineers and programmers will be well defined, and the problem of proper documentation would be significantly eased.

High level languages claiming to be suited for real-time applications should be measured, among other factors, against the demands listed above. Several languages have been introduced in recent years by members of the NATO community, and the more important were examined by the Working Group. They are:

- JOVIAL (USA)
- LTR (France)
- PEARL (Germany)
- CORAL 66 (UK)

and they are defined in Annexes G to L.

Following the description of these languages, an attempt was made to establish a comparison. A questionnaire was composed, consisting of 44 questions which were addressed to each of the languages. This assessment is shown in Annex M.

3. US EXPERIENCE

3.1 Introduction

This report on the digital avionics efforts in the US Air Force is intended to convey the status of ongoing projects in the area, experience derived from current programs and will show proposed future plans. The digital avionics concept has been accepted as the way of integrating weapons systems of the future and is hinged on the data bus concept with standard digital interfaces. Growing from that into standard computers, controls and displays, sensors and subsystems will reap the benefits of lower life cycle costs. It must be remembered that a digitally integrated weapons system is a software integrated system and this requires that heavy emphasis is placed on software performance and reliability. This report will provide the following information:

- (a) A justification for the need of standardization in digital avionics.
- (b) Multiplexed data bus experience in the avionic weapon systems.
- (c) Higher Order Language (HOL) experience in avionic weapon systems.

The information will show that the digital avionic integration approach is based on the time-sharing of data paths (i.e., data bussing), the liberal use of avionic computers and data processors, the sharing of data entry and display devices, and the use of avionic sensors with standard interfaces.

3.2 A Justification for the Need of Standardization in Digital Avionics

When integrating the avionic system, often called "architecting", the data is generated and consumed by sensors, transported via multiplexed data busses and processed by computers. These digital avionic architectures can be simulated, based on information flow in real-time large ground based computers and on dynamic "hot benches", prior to any avionic procurement, in order to select the most cost effective approach. The reason that this can be done is that the integration of the hardware is done to standard signal interfaces and the information flow control of the data and integrating algorithms are done in software. Software controls the data flow, its timeliness and the accuracy of the solutions. The software implemented operational algorithms can be exercised by dynamically supplying the required data, through real or simulated inputs, and monitoring the outputs. Reaction to changes in data flow (i.e., data degradation and/or failure) can be exercised. In other words, the software analyst/programmer now also becomes the system integrator, the systems architect.

The impact of digital avionics is great; its proper application to avionic systems can provide increased performance, lower hardware cost (through standardization), increased reliability, less proliferation, reduced operational and maintenance costs and eventually reduced acquisition costs. Savings can be even greater if one could learn how to harness this

digital monster. The time to act is now. There is so much to do. The complexity of the problem can be reduced through standardization; however, the approach must be orderly. In digital designs, unlike that of its analogue counterpart, there are many more ways of implementing the same function in a different manner through clever use of logic, computers and software and making the various unique designs perform the desired function correctly. To standardize on fixed hardware designs would be stifling ingenuity and progress as well as technology. To standardize on clearly defined, neatly partitioned functions and to standardize on signal interfaces is the solution. Compatibility of hardware functions to be integrated is a necessity; commonality of hardware is desired within and between weapons systems. In the area of digital avionic integration (i.e., integration based on a data bus concept) certain building blocks can be standardized. They are the multiplexed data bus, computers, software, controls/displays, subsystem interfaces and some common general purpose sensors. The reason that the above items can be standardized is that in the integration of these items (building blocks), the systems engineer, or architect, can analyze the systems problem and "program" these shareable multipurpose devices to give him the necessary solution. The key is that simplicity and low cost is stressed in standardization. It means that a standard item shall be designed not to satisfy the peculiar or unique nor the super high speed requirement but the *general* requirement. If 70 to 95% of the requirements can be met with the standard item, there is sufficient payoff to make it worthwhile; i.e., when the knee in the curve is reached where cost, complexity, etc., go up sharply without significant performance gains being realized, it is time to stop and to include more.

The logical approach and order to standardization in digital avionics is as follows:

- (a) Digital and Discrete Interfaces,
- (b) Multiplexed Data Busses (Serial and Parallel),
- (c) Airborne Computers,
- (d) Avionic Software (Algorithms, Programs, Support Software, Higher Order Language(s) (HOL)),
- (e) Shareable Digital Controls/Displays,
- (f) Common, Functional Sensors.

The building block approach to digital avionic system design, its simulation, its dynamic reaction and its performance is based on clearly understanding the data flow, the data source accuracy and update rate, A/D conversion, transmission and processing errors and the processing algorithms used. This requires that there be absolutely no doubt of sensor data (signal) characteristics at the back end of the sensor (i.e., at its output). It's the back end of the sensor that must be integrated into the computer, not the front end. In the past we took "pot luck" and received the data definition and characteristics usually too late and often they were different from the original specification. To integrate these incompatible sensors with the computer becomes a last minute ordeal with minimum engineering being put into the converter/interface design. No wonder the converter box is the greatest data error source both in accuracy and timing. There is no doubt that the "Standard Digital Interface" is the key to success for digital avionics. With or without a data bus, it is a must if any NATO benefits are to be obtained.

It is, therefore, easy to understand why the data bus has been so widely accepted; it provides the standard interface inherently. A standard data bus really provides two standard interfaces. One is between bus electronics and the subsystems (it can be standard no matter what bus concept or transmission media is used) and the other is at the bus medium itself (at the twisted pair connector if MIL-STD-1553A is used). If the second interface is used, the bus electronics can be built into the subsystem. This can save considerably on connectors and harnesses between subsystem and the data bus electronics box as well as eliminating the need for a special housing, power supply, etc., for the bus electronics. The choice is up to the systems architect. Another consequence of the bus concept is that each avionic subsystem must do its own analogue-to-digital and digital-to-analogue conversion. This does away with the central converter in the integration (central computer(s) and make them standardizable. (In the past it was the unique input/output (conversion and interface) requirements that made the same computer used in different applications hardware incompatible.)

In order to standardize on airborne computers, one must realize the broad range of applications that digital computers are used for in current weapons systems. Despite the flexibility of software, there still is a need for sizing the computer to the job. Therefore, it is recommended that a limited family of computers be selected for standardization. There is a need for a mini, a medium and a large computer, measured by capability, not by physical size. Each of these computers shall have a defined instruction set, sized to its individual capability, but upward compatible. It is realized that a defined instruction set can hinder a hardware designer's creativity, but in reality it should provide a challenge to them to live within the constraints and still provide the competitive product.

Microprocessors are coming into their own now and are presently widely applied to sensors by designing them directly into hardware to replace digital hardwired logic. Programmable Read-Only-Memories (PROMs) are generally applied here. Because of this and because industry is introducing new chips almost daily, it appears that it may be a little premature to start standardizing microprocessors.

After standardizing digital interfaces, data busses and computers, the next task to be undertaken is the standardization of software. Three distinct efforts are involved here: the development of the operational software, the development and use of the support software, and the maintenance of software programs in the field. Algorithms can be standardized and, in some instances, coded programs could be if standard computer instruction sets are used. A standard higher order language (HOL) is much more likely if computers are standard because one optimum compiler can be developed and refined through the application of software-identical airborne target machines. Other support software (simulation, test,

etc.) should become common, too. So it is only logical that the software support workload in the field would be reduced and software reliability would go up.

In the cockpit the pilot and/or operator who interfaces with the system communicates through pushbuttons and switches and reads lights, dials and numeric displays. We all know that the cockpit is crowded with single function devices and that it is becoming a real human engineering problem to operate the weapons systems. In all that noise there exists valid, timely information required to perform the mission, and the operator most likely could become more effective if multipurpose keyboards and multifunctional displays are used and only pertinent information is displayed. Again, it is the computer software that interprets buttons pushed and it draws and positions symbology on the CRT. Therefore, digital controls and displays are an ideal candidate for standardization. Multifunctional devices with standard digital bus interfaces are vital parts of this building block approach to systems architecting.

Finally, sensors such as radars, TACANs, radios, air data computers and the like are the primary sensor building blocks that can be standardized, at least in part. What is meant by that? Sensors most likely will be unique in themselves. They actually provide a needed function. They are a source or sink for data to the systems architect. To the data bus, their signal interface must be standard and their functional performance must be definable by the data they generate or use. In mission-oriented sensors one expects little physical hardware commonality because of the continued attempt to improve sensor performance. In the navigation, flight and communications area there is a high potential of common hardware usage (Inertial Measurement Units, TACANs, UHF/VHF radios, altimeters and the like). For example, the same sensors might be used in all weapons systems designed over a lengthy time period and changed only if some significant breakthrough occurs that surmounts a deficiency (i.e., performance, reliability, size, cost, etc.) and then this new design would become the standard.

If standardization occurs in all the above areas and real "value engineering" principles are applied to the systems design process, then some great life cycle cost benefits can be obtained. Systems architecting will be one of challenge and compromise with what may seem endless trade-offs, overdesigns and underdesigns for the sake of commonality, a real test of self-constraint to avoid the use and development of "new and unknown" technologies, learning how to say "no" to the whims of others and to your own, and designing the system to be a whole, not the sum of its parts. If this type of self-discipline will come about or not, only time can tell.

3.3 Multiplex Data Bus Experience

Multiplex data bus experience that has been encountered in Air Force weapons systems has been extremely encouraging. In the *F-15 mission avionics integration*, several redundant twisted pair busses exist for reliability's sake. (Fig.3.1). There is a full dual redundant primary bus system under central computer control, a dual redundant degraded (not to all sensors) bus system under radar computer control, and then finally some emergency hardwire backup interconnections for a few sensors. In case of primary bus failure, the bus system automatically switches to its standby alternate. Except when initially tested on the hot bench, where failures were purposely induced, and random I.C. failures, the system has never had a failure attributable to the bus concept. This includes avionic flight tests to date. The reliability of the bus technique has been extremely high. When bus failures occur, they are expected to be either a non-functioning computer or battle damage. In the B-1 (Fig.3.2), before F-15 test data was extensive enough, the EMUX (electrical power control) system was made quad redundant with emergency condition hardwire fallback in case of total EMUX inoperability. Problems have been primarily in the programming of its computers and electronic hardware failure, not bus technique. Current data is encouraging; the electronic power people have progressed on the learning curve of digital avionics (they previously were totally analogue/relay oriented). If it had to be done again, redundancy would be used less, maybe just dual. The reason given is that the current design has lower reliability because of the larger amounts of electronics. The AMUX (Avionics Multiplexed Data Bus) design is working fine on the hot bench.

The next data bussed weapons system endeavour is the F-16 which will be designed to the tri-service MIL-STD-1553A, (Annex B). Lessons learned will be applied to that design. Many US contractors have built hardware to MIL-STD-1553A and all report greater data transmission reliability and lots of flexibility. Some use terminals with micro-processor control that can be reprogrammed to satisfy a variety of users. Some other users of the data bussing concept are NASA's Space Shuttle, the Navy's F-18, AFAL's DAIS, Canada's LARPA, some RPVs and many Laboratory programs. As for the USA, multiplexed data busses are a reality. They are accepted and no longer have to be proven.

3.4 Higher Order Language Experience

In the B-1 central computers a higher order language is being used. It is J3B, a JOVIAL derivative (Annex H). Initially, it was a language specification without the existence of a compiler for the Kearfott SKC-2000 computer. Other support software was written in FORTRAN. As part of the DAIS program an attempt was made to do an Air Force wide investigation of HOLs. The result was a committee generated definition of an avionic oriented HOL called J73/1 (Annex G). A language compiler targeted at the DAIS selected Westinghouse computer is under development. The uniqueness of the J73/1 language is that it is written in itself; therefore, no second compiler language is needed to maintain it. J3B maintenance is done in AED, a company peculiar language. Sole source and company proprietary problems can arise in such a situation. B-1 experience has shown that a HOL can be helpful in the development and checkout of the operational flight programs; however, it appears it does not solve the problem of overloading a central computer

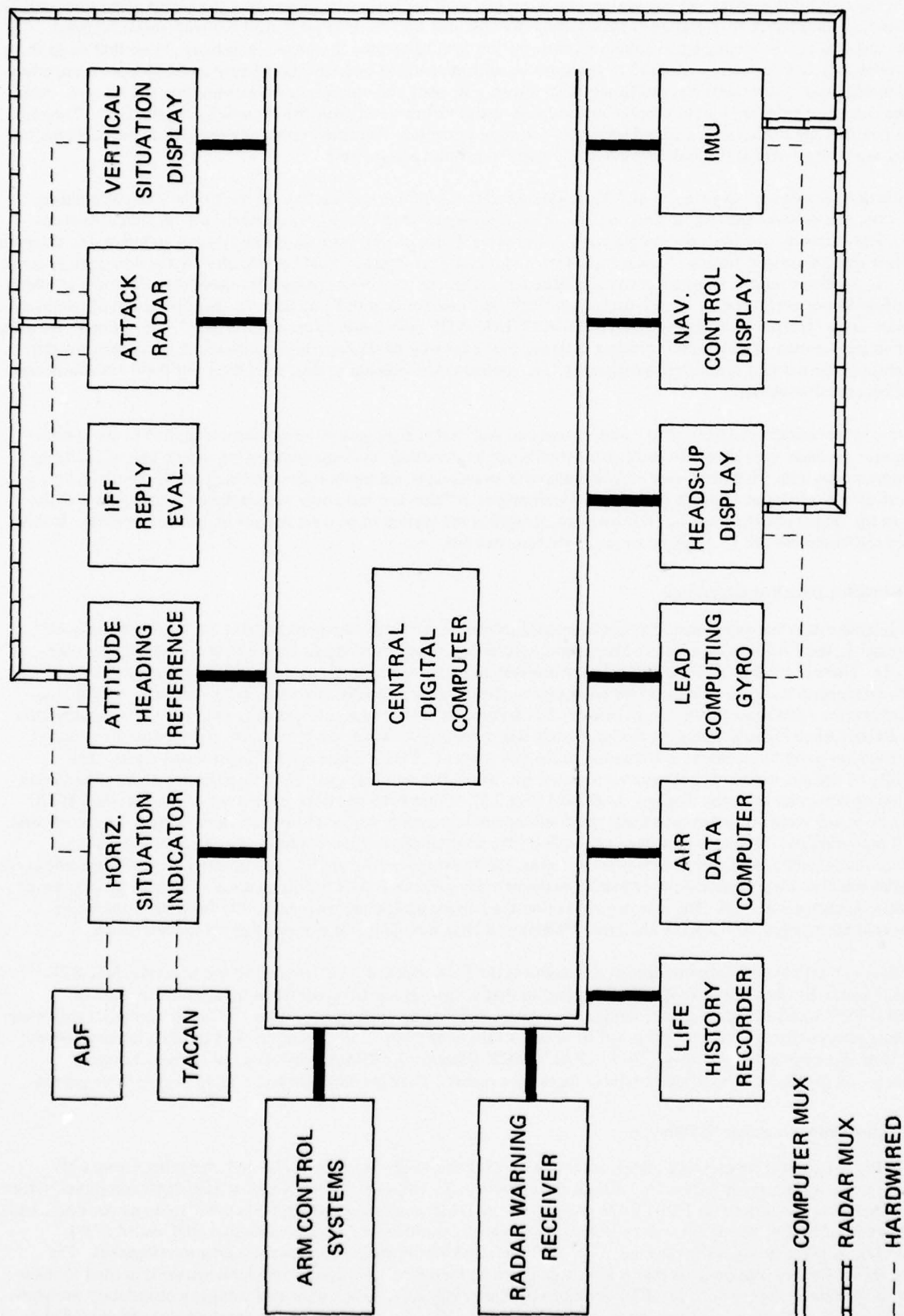


Fig.3.1 F-15 digital system architecture

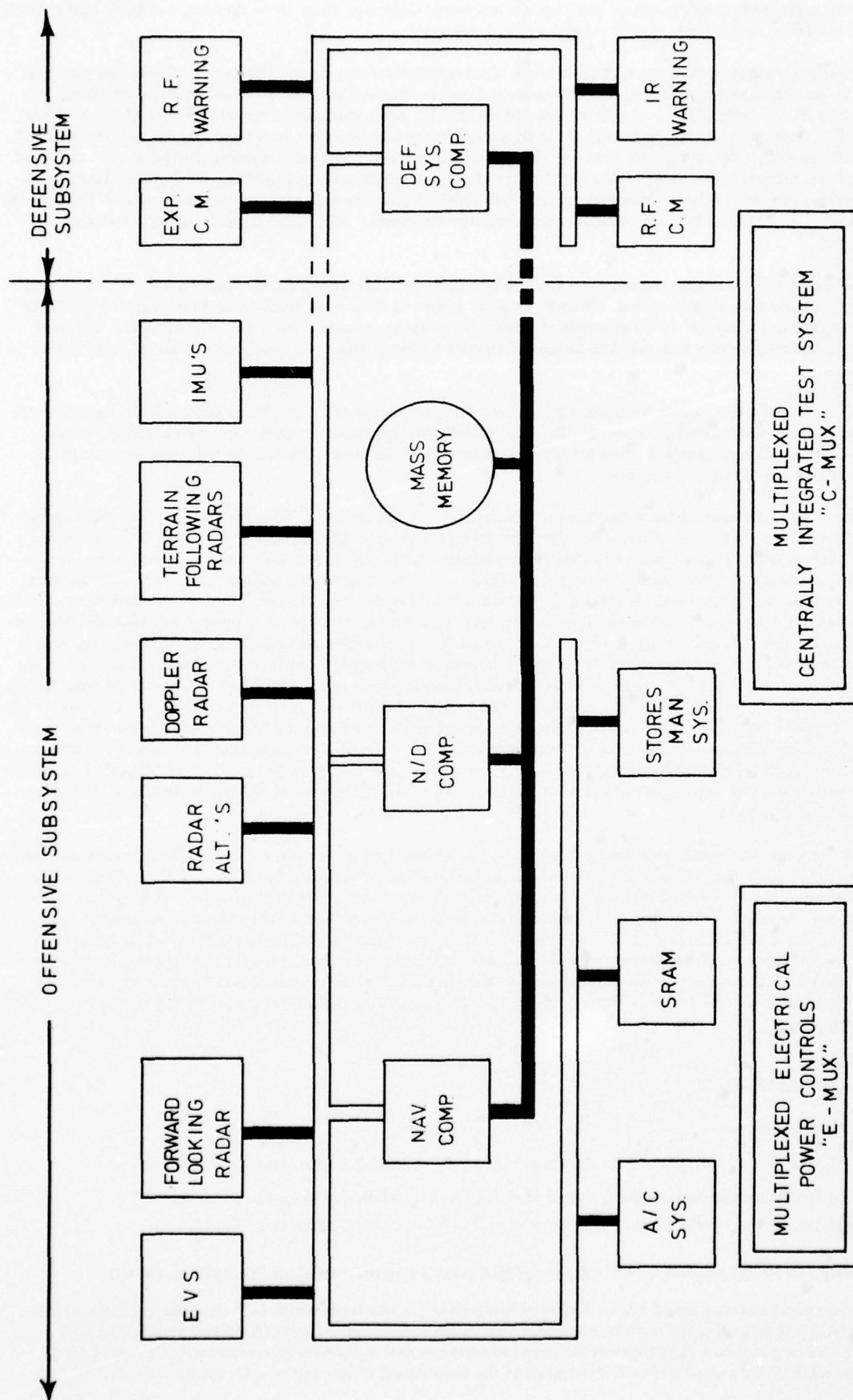


Fig.3.2 B-1 digital system architecture

complex with software that is fragmented and highly interleaved. Currently, there are again a series of HOL studies being performed in the Air Force, Navy, Army and also on the DoD level.

A higher order language is an attempt to decouple the programmer from the mechanics of assembly language level machine characteristics and to automate such things as address bookkeeping and mathematical equation decoding. In other words, a HOL can interpret English level task statements. However, this poses a problem in that when one establishes the HOL statements, they must be written in the jargon of the user (avionics, command and control, testing, etc.). This gives rise to a requirement of more than one HOL, maybe one for each uniquely oriented discipline. Another need arises in systems using real-time control; the input/output task statements must be in addressable assembly language to enable real-time sampling of data. Consequently, this establishes the requirement that any weapons systems' HOL should be capable of accepting both HOL and assembly level language statements. J73/1 does permit this type of statement interleaving.

It is a known fact that HOL compilers are less efficient than their assembly language counterparts. Ten to twenty percent in speed and code is often stated. Efficiency can be improved if the same HOL could be used for a longer time and on more than one program. In other words, it is task and computer oriented. So if one can define the "avionics" task to be used in conjunction with standard airborne computers, then it should be worthwhile to invest in compiler improvements.

With the current trend toward federated computer architectures some of our problems are eased. For example, the central management computers have no longer the highly interleaved programs that the multitask central computers usually have. The efficiency problem should not be a problem any more either, because the federated architecture is based on cheaper computers and memories.

The primary advantage of HOL is that the programmer can become independent of the hardware peculiarities and can, therefore, devote his time to efficient algorithm implementation and proper timing interactions. One interesting result of a study was that programmer efficiency did not improve with HOL usage. Independent groups were set up to program and debug a benchmark problem, one group in HOL, the other in assembly language. After they got them to execute, they repeated the process, but this time they switched languages. Performance, length and execution time of the coded programs and programmer coding and debug time were recorded in each case. The result was surprising. The performance and relative efficiency of the code and the programmer's time were independent of the language, they were a function of the ability of the programmer. One other observation was noted. Consistent with each programmer, HOL was more quickly coded and took longer to debug; assembly language was just the opposite. Oddly enough, total times were roughly about the same for any single programmer no matter what software language was assigned. In other words, HOL cannot make a bad programmer good! HOL can, however, make operational software easier to track during development, it is more maintainable in the field and provides much better overall documentation. Changes can be incorporated via recompiling rather than patching. HOL will permit the systems analyst to put his thoughts directly into code which reduces the analyst-to-coder communication problem. Therefore, HOL is good and will be applied in all future Air Force weapons systems.

In summary, the Air Force supports digital avionics, but digital alone does not save money. Digital avionics, however, can be much more easily standardized and reprogrammed for multiple applications. They, therefore, can be procured in larger numbers, more emphasis can be put into their development for increased reliability, and logistic support can be reduced. All in all, it eases the avionics proliferation problem, lowers the hardware cost and reduces the spares inventory, support training and skill level requirements. If the buying community gets larger than the Air Force (i.e., Navy, Army, NASA and NATO countries), then additional cost savings can accrue. Finally, if the standard interfaces and MUX bus integration concepts were standardized within NATO, international buys could be made on standard hardware items. The systems design (architecture) and the software system that makes it operate can still be the buying nation's own.

4. FRENCH EXPERIENCE

4.1 Multiplex Data Busses

Two documents concerning multiplex data busses are given as Annexes C and D, and are as follows:

- Provisional recommendations for a system of multiplexed transmissions along bus lines (see 4.1.1).
- Definition of the digibus for the exchange of digital data in a combat aircraft (see 4.1.2).

4.1.1 *Provisional Recommendations for a System of Multiplexed Transmission along Bus Lines in Aircraft*

This document has been issued by the Technical Integration Committee (Comité Technique Intégration), which is a Working Group representing the services, pilots, manufacturers and equipment users (Air Force staff and aircraft companies). Their aim is to collect together views and experiences within the field of system integration in order to define, if possible, French standards or to participate in the formulation of international standards.

Since multiplex data bus techniques are one of the aids to system integration, they were among the first preoccupations of the Technical Integration Committee.

As their name indicates, the 'Provisional Recommendations' are neither a specification nor a description of the system, but, rather, a compilation of those systems which are technically viable, together with an indication of the options between which it is difficult to decide for the definition of a system until sufficient flight trials have been carried out. Moreover, certain sections have been left in abeyance (for example, that on screening), since these can only be written with a thorough knowledge of the facts when experimental results are available.

These recommendations will be revised and will be complemented by experiments insofar as they affect one or other of the specifications for a particular field of application.

Henceforth, they must be adhered to for all new French data bus studies.

Several series of experiments or studies are being carried out; for example:

- a study of digital transmission on board a transport aircraft (avionic data bus), with flight trials to take place in 1976;
- a study of digital transmission on board a military aircraft (avionic data bus), described in 4.1.2 and Annex D;
- a study of an electrical network control bus, with experiments to be carried out during 1976;
- a study of an intercomm sub-system, with flight trials during 1976.

Thus 1976 will provide a great amount of data on the *in-flight* behaviour of data bus systems.

4.1.2 Digital Data Busses for Combat Aircraft

4.1.2.1 General

This data bus is the result of studies which have been carried out over several years. A version designed for the Super Mirage is being developed; ground and flight tests will be carried out during the next few months. At present, this is basically an avionic data bus, connecting together the major components of the aircraft nav/attack system.

In this data bus, failure is guarded against by providing redundancy:

- in the bus coupler, to avoid line breakages and abnormalities caused by mal-functioning of the items connected to the data bus,
- in the control units, to try to avoid electronic breakdowns in these units.

Transmission is serial at 1Mbit/sec, with return-to-zero biphase modulation, in 10-bit characters.

The development which is now in progress will use standard couplers (COS) in the form of a single card, the same card being used in all items of equipment connected to the bus. At the level of the interfaces between individual equipments and the bus, this approach leads to a standardization, not only functional (signal specification) but also physical, by the use of a common sub-assembly, which leads to homogeneity in design and a requirement for longer production runs, with a consequent effect on cost.

4.1.2.2 Applications

The use of this bus is envisaged on all future military aircraft avionic systems - Super Mirage, F1M53, Atlantic Mk 2, etc.

On the Super Mirage, the bus, which connects the major components of the nav/attack system, operates in two ways:

- a primary mode, using the main computer, which is also used for bombing computations and failure assessments,
- a reversionary mode, using the computer associated with the inertial platform.

4.1.3 Advantages of Digital Data Busses

From the digital bus systems France expects to realise certain advantages, including:

- decrease in weight and volume of cables;
- decrease in the complexity of the interconnections; in fact, certain complex systems may not be viable at all without the data bus technique;
- flexibility of adaption to different system configurations, due to the modular concept;
- flexibility of system modification, allowing the addition or replacement of an item without important re-wiring;

- standardization of the interfaces between items, which leads to standardization of the items themselves, with a beneficial effect on costs;
- a homogeneous structure, with the items becoming more and more digital. In this connection it should be noted that the concept of data exchange by bus between the various sub-systems has been used for a long time within computers, for the obvious reason of modularity;
- fault diagnosis: the control unit can, in flight, measure certain operating parameters and, in case of anomaly, record them for later ground analysis;
- aid to maintenance: by simply connecting an external system to the data bus, access is assured to most of the aircraft equipment.

4.2 High Level Languages

4.2.1 LTR Presentation

The high level language LTR (Language Temps Réel – Real Time Language, Annex I) is standardized in France for military applications in real time.

It was developed initially for the integrated control of armament systems on board surface ships for the French services.

The language, which is described in detail in Annex I, is particularly well-suited to real time problems (control of tasks and resources, bit manipulation, etc.) and can be used on small computers.

Efficient compilers have been written, having an expansion factor of about 1.3. In this context, the expansion factor is defined as the number of machine instructions generated by the compiler for any given application, to the number of machine instructions obtained by programming the same application in assembly language.

This expansion factor, then, defines the necessary increase in memory requirement, as well as the approximate increase in time to execute the program.

4.2.2 Applications

Up to now, the language has been used mainly by the Navy and Army. In the aeronautical field its use is envisaged for the Bréguet Atlantic Mk 2 (anti-submarine patrol aircraft).

It is envisaged that its use will be extended, progressively, to various other aircraft projects. This use is particularly connected with the existence of compilers for the various computers used, and several are either available now or are in the course of development.

4.2.3 Advantages of High Level Languages

High level languages have several advantages:

- programming of an application in high level language leaves the programmer free to take a greater interest in the problem to be solved than in the computer;
- it is relatively easier for the system designer to undertake the programming;
- a program is more readily understood by more than one person;
- the programs obtained are 'transportable' from one computer to another, with the same computer logic used in each case;
- the time and cost of development and maintenance of a program are lower.

These advantages must be set against the fact that the required memory volume and, hence, its cost are greater because of the expansion factor. The overall advantage, however, increases with time, given that manpower costs are rising whilst memory costs are decreasing.

All these advantages would, however, be lost if high level languages were allowed to proliferate. It would then be necessary to increase the number of compilers, which are costly and complex programs, difficult to develop, and the advantage of transportability would be lost.

It is, then, very desirable to avoid modifications to existing languages since the technical advantages which could accrue are very likely to be outweighed by the inconvenience due to incompatibilities, which are sure to appear.

5. GERMAN EXPERIENCE

5.1 The German Experimental Guidance and Control Program for Helicopters (HSF)

5.1.1 Background

The armed forces in Germany are increasingly plagued by the shortcomings of the present helicopter guidance and control systems. The systems are not fully adequate to meet the military requirements under the weather conditions prevailing in central Europe.

The equipment required for the execution of bad weather operations for helicopters was found to be largely developed; investigations revealed, however, that by adding equipment piecemeal, systems of unjustified complexity would result. It was believed that the problem has to be addressed in the direction of new system architectures rather than in the development of new sophisticated equipment.

The HSF experimental program to verify the performance and integrity of newly architected guidance and control systems was consequently started more than a year ago. The program is directed by the German MOD and the work is subcontracted to a variety of German companies and research institutes.

5.1.2 The Problems

In principle to-day, guidance and control equipment is providing increased capability and performance towards bad-weather capability for helicopters. If, however, the systems are architected simply by adding equipment piecemeal then the required reliability cannot be achieved, in many cases, unless excessive redundancy is applied. Sophisticated guidance and control systems architected by applying redundancy on the equipment level are always costly. Lessons learned from this method of system architecture also show evidence of problems of maintainability and operational integrity. Under some circumstances equipment functions can even contradict each other during different parts of the mission, thus increasing the pilot workload for coordination.

Digital techniques appear to offer the means to achieve the necessary improvements in capability and performance as well as in reliability and integrity, not just at the equipment level but also at the system level. What this means, in fact, is that all or part of the sensors and subsystems are no longer functionally autonomous or isolated as in present systems. The consequence of the new approach for system architecture is, therefore, the necessity to interface all equipment parts of a system in an appropriate manner. Consequently, one of the main aims of the HSF program is to seek effective means to interface, both in hardware and software terms, the equipment and subsystems of a guidance and control system in a hierarchical real-time system where functions can be shared by different equipment.

The specifications for the HSF program require that equipment from different suppliers shall be used, and that new equipment can be added later. This means that standards for a serial multiplex data bus have to be selected. The preliminary specifications of the proposed multiplex data bus which has been selected are given in Annex E. Further specifications within the program call for the partitioning of submodules as processors, converters, memories, and so on. The appropriate approach to these requirements seemed to be a parallel multiplex bus which allows for the use of supplier-independent, line-replaceable submodules. Specifications for mechanical dimensions, plugs and sockets, given in Annex E, are required for the parallel multiplex bus in addition to the functional rules.

Present digital guidance and control systems are based on the use of computers and processors and, consequently, must rely as much on appropriate software architectures as on hardware architectural schemes. Standard interfaces and multiplex data busses provide for the ability to interchange or add new equipment and sensors during the life-time of the system without excessive rewiring of the aircraft. As the architecture of new guidance and control systems implies that functions and signals are shared, additions or changes to equipment and sensors also implies, as a rule, software changes and reprogramming. Reprogramming or adding new software modules is cumbersome and tedious without an agreed standardized structure for programming and linking software modules. Within the frame of the experimental guidance and control program for helicopters (HSF) attempts are therefore being made to tackle the problems.

Contrary to the hardware systems aspects, the problems of portable software are considered, from a practical viewpoint, as long-term studies. What seems to be needed is a software language with a rigorous technique for syntactically describing a complete system. This includes the syntax description of the hardware configuration, the hardware interfaces, the tasking and scheduling and the real-time operating system functions required for guidance and control systems. PEARL (Process Experiment and Automation Real-Time Language), which has been developed in Germany as a real-time process language, under the auspices of the Ministry of Technology, could be suitable for this purpose, provided its implementation for on-board computers and distributed multiprocessor systems proves to be feasible. Consequently, PEARL has been selected as a candidate to be used in the second stage of the HSF. A short description of PEARL is given in Annex J.

5.1.3 Mission Requirements for the Demonstration of the System

Several missions have been analysed to specify preliminary guidance and control system performance and capabilities

required for bad weather conditions. Transport missions have been selected to demonstrate the feasibility of the guidance and control system, and these missions include tactical transport, logistic transport and utility transport.

According to Figure 5.1, the experimental guidance and control system shall demonstrate the capabilities for take-off and landing missions on

- Normal approach or take-off at sites with installed IFR or GCA ground equipment.
- Approach or take-off at field sites with some communication aids and field equipment installed for guidance.
- Approach or take-off at sites without ground equipment. (Only the geographical coordinates and the geographical conditions are known.)

Take-off and landing shall be flown using normal procedures for visual flight conditions VFR or for IFR (GCA) conditions. Possible new procedures which are more convenient under conditions of conflict will be investigated in later stages of the program. The SETAC system, a mobile ground equipment to provide azimuth, elevation and distance information, will be used as a landing aid.

The transport missions specified in Figure 5.1 require normal en-route navigation under ATC and flights with ground-independent navigation. Provisions for ground-independent navigation and terrain-following are required in later stages of the program. The requirements for the contour flights are 30–90 m above ground level.

5.1.4 System Philosophy

The manual control workload for the helicopter puts too heavy a burden on the pilot during such parts of military missions as:

- bad weather and night flight;
- low level, contour or formation flight;
- flight under threat conditions.

The present stability augmentation systems and autopilots alleviate the manual control problem; however, the autopilot is not fully adequate for the tactical guidance of a helicopter with rapidly changing manoeuvre demands. The stability augmentation systems, on the other hand, do not have, in many cases, the desired performance for good handling capabilities during the whole flight regime. In addition, the authority of both systems must usually be limited for reasons of reliability.

Investigations have indicated that major improvements could be achieved by more sophisticated manoeuvre demand systems, where the disturbances would be automatically controlled and the individual movement would be controlled directly by the pilot. The selected system coordinates the four controls as to the following:

- velocity demand for three translatory directions, horizon-oriented with respect to the longitudinal axis of the helicopter (geodetic coordinates),
- angular velocity in the yaw axis, while hovering, or rate for a normal, coordinated turn.

This manoeuvre demand system has a hierarchical architecture with different reliability on different levels.

- (1) On the lower level with respect to the functional capability a minimal redundant system provides acceleration demand for the three translatory directions and rate demand for yawing or normal turn.
- (2) On the upper level the nominal system provides the desired velocity demand for the three translatory directions.

The minimal system will be triplex to achieve the required safety. The predicted failure rate is of the order of 3×10^{-9} failures/hour. The nominal system is partly duplicated; duplication, however, is achieved with no large extra cost because the equipment required functionally is shared with the rest of the guidance system.

The guidance information for the required missions, i.e.:

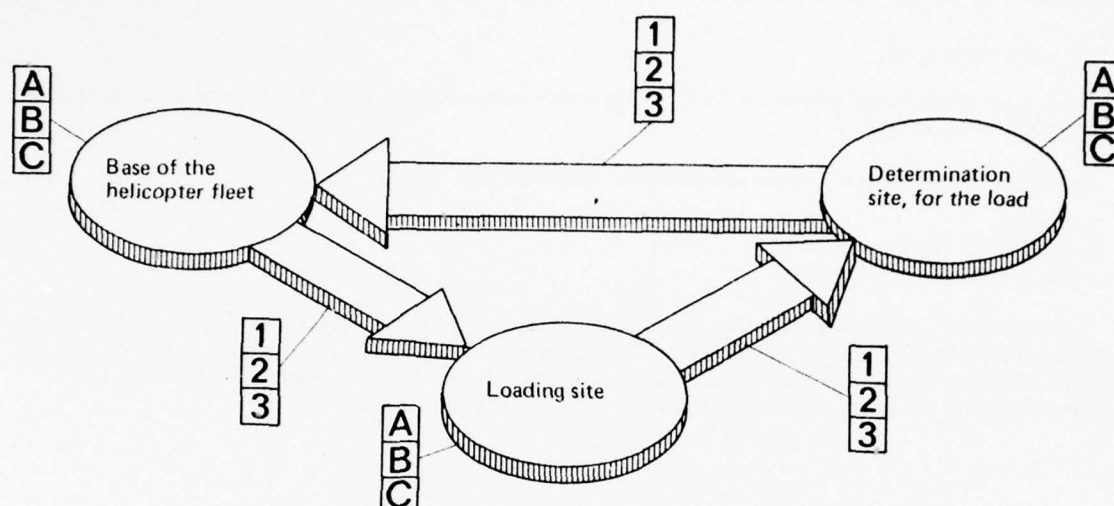
- en-route navigation under military IFR control or ATC;
- flight independent of ground-based navigational aids;
- low level flight with ground-independent navigation including terrain following or avoidance in later stages of the program,

is achieved with::

- air-data information;
- doppler velocity information;
- inertial sensor information, and
- Tacan information.

Missions

Tactical	Transport
Logistic	Transport
Utility	Transport



Conditions

En - route flight

1	According ATC	300 m (600 m) AHO $r = 8$ km
2	Independent of ground - based navigational aids	150 m AHO
3	Ground-independent with terrain following	Contour flight 30 - 90 m AGL

Take off and Landing at the site

- A** IFR or GCA conditions
Helicopter minima 0,5 km 60 m
Navigation and communication aids
Lighting aids
- B** Communication and lighting aids only
- C** No ground equipment. The landing site has been fixed and explored.
The geographical coordinates are the only information for the helicopter.

Fig.5.1 Transport mission for the experimental guidance and control program for helicopters

In later stages it is planned to investigate the use of additional optoelectronic equipment, such as FLIR or LLTV, and terrain-following or terrain-avoidance radars.

An on-board computer is used to process the different sensor information and to make the best use of it. Guidance information is presented to the pilot via an electronic vertical and horizontal situation display. Modes can be selected by the pilot. In the case of a failure of one display, the remaining display will be used for both vertical and horizontal information display. All essential signals for processing the required guidance information are generated duplex within different sensors. In the case of a failure in the on-board computer, limited guidance information processing is still available since the required functions are shared with one of the data processors of the manoeuvre-demand system.

5.1.5 System Architecture

The mission requirements outlined in 5.1.3 and the system philosophy described in 5.1.4 lead on to the system architecture shown in Figure 5.2.

In technical terms, the system must comprise three essential parts:

- sensors,
- control and display,
- signal processing.

The experimental guidance and control system must be hierarchical, with different reliability at different levels. At the higher level, the on-board computer system coordinates all the complex functions to achieve the fully specified system performance. At lower levels, command augmentation (CAS) and stability augmentation (SAS) functions, minimum display functions and others are performed within the safety margin required.

5.1.5.1 Sensors

Within the present experimental system, a triplex system has been selected to achieve the required reliability of the CAS. As the minimum system comprises a translatory acceleration demand system, triplicated inertial acceleration signals are required for control and attitude and rate signals and also as feedback. The triplicated signals are generated within a simplex precision strap-down system, the conventional horizontal and vertical gyros aboard the helicopter and additional low precision inertial instruments (Fig.5.2).

Guidance information for self-contained area navigation is provided mainly by a doppler-inertial system, signals being generated with the doppler radar equipment and a strap-down inertial platform. The air-data sensor system also generates coarse guidance information for self-contained navigation, which is used for comparison and as back-up in case of a failure. Flight instruments and a Tacan system yield the signals for en-route navigation. Amendments to the Tacan equipment provide distance, elevation and azimuth information in conjunction with the mobile landing aid Setac, during the approach to field sites.

During later stages of the program, as mentioned above, the intention is to evaluate the use of optoelectronic sensors such as FLIR and LLTV as well as terrain-avoidance and terrain-following radars.

5.1.5.2 Control and display

Two electronic displays are used to provide the pilot with the guidance and command information needed for the mission. The use of cursive displays and of raster displays will be investigated during the program.

As it seems important that the pilot has a full picture of the actual state of the guidance and control system, rather sophisticated control panels are being used. The pilot can select modes, activate certain tests via the control panel on one side, and gets a presentation of the system state (such as a degradation message in case of a failure) on the other side.

Normal controls, stick, pedal and collective pitch are maintained as a safety system in the helicopter during the experimental program. The missions during the program are flown with a side-stick system, however, where the pilot commands manoeuvres. The side-stick system has triplicated pick-offs.

5.1.5.3 Signal processing

The guidance information is generated in the sensors and presented to the pilot mainly via the display system. As the requirements dictate duplication of the main guidance functions, the data multiplex bus connecting the sensor with the display subsystem must be duplicated also.

The subsystems are linked to both serial multiplex data busses via standardized interfaces. The subsystems themselves can have their own data processing unit and the coordination of the subsystems and part of the central processing is done within the on-board computer system. No decision has yet been made as to whether the on-board computer should be duplicated, or whether, in the case of failure, one of the decentralized processors should take over central control and provide the data processing capability required for the display of minimum guidance functions.

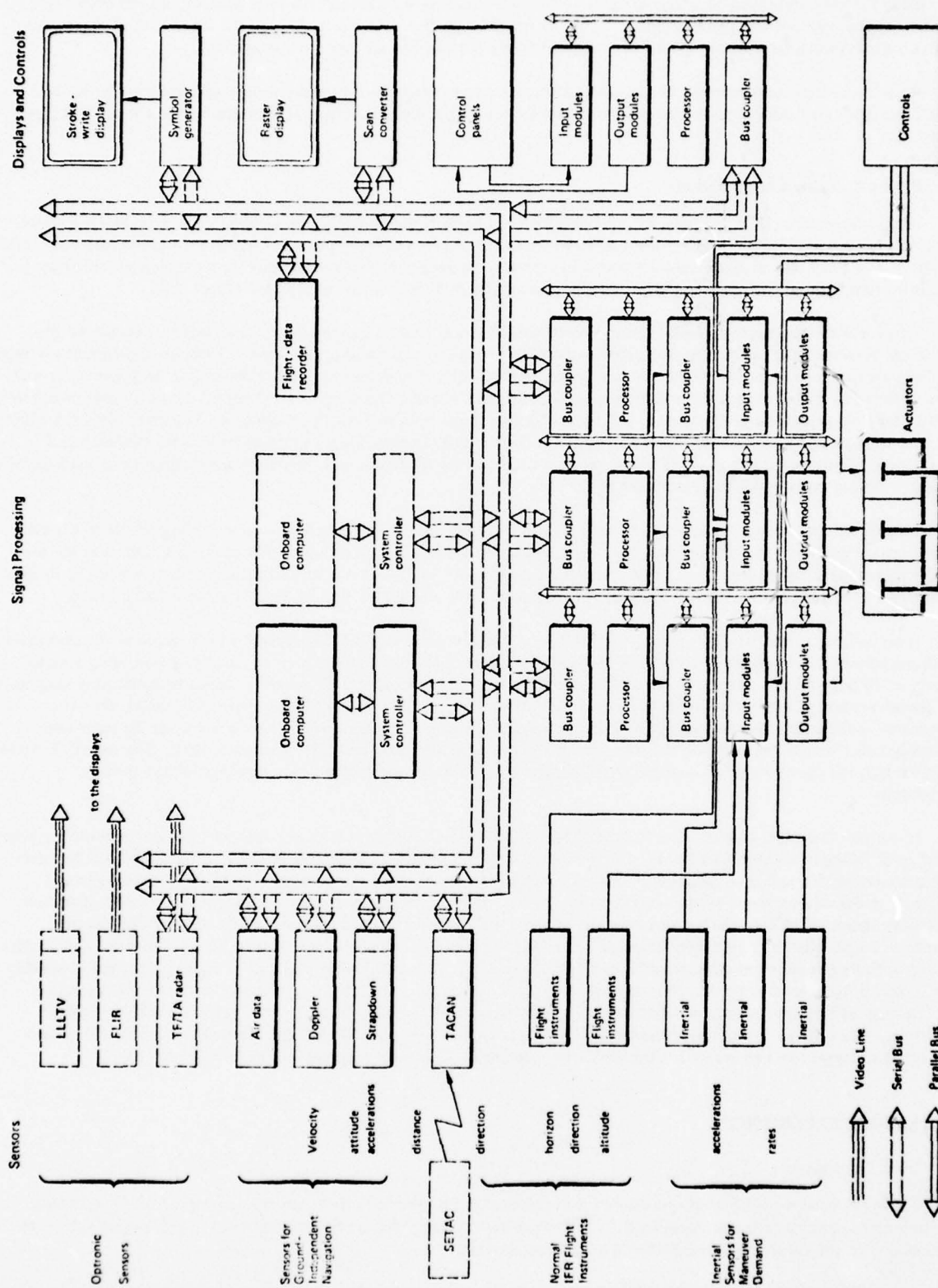


Fig.5.2 Architecture of the experimental guidance and control system for helicopters (HSF)

The manoeuvre demand system is triplicated. Each simplex system is connected to the main sensor system and to the on-board computer as well as to the control panel via the duplicated serial busses. In the experimental HSF system some inertial sensors and the actuators are connected by dedicated lines. The main signals of the three lines of the triplicated system are compared to check whether any failure could have occurred. No final decision has yet been made as to whether the required comparison signals should be fed through the serial bus system to the three lines of the triplex flight control system, or whether special bus couplers to the parallel bus are more advantageous.

Since many data conversions and logical decisions have to be made for the signal supply to and from the control panel, and since partitioning of the modules was a design aim, a parallel bus system is provided within the control panel subsystem.

5.2 Future Software Considerations

Within the present HSF program, no special efforts are being made to apply new software concepts like high-level languages for process control, standardized software modules and formal descriptions of hardware architecture and configuration. For example, all algorithms are coded in assembly language. Real-time features are implemented either by calling on functions of the operating system or by planting them directly into the relevant object code.

This software approach to digital computerised guidance and control systems is not satisfactory, because all the interfaces between design engineers and programmers, and between individual programmers working on different software modules are designed more or less independently and individually. Problems arise during integration in proving the software correctness and in proper documentation, and during the course of later software changes when different members of staff take over work on the programs. All these problems have been universally recognized. The main aim of the HSF program in the early stages is to prove the function of the modular hardware concept with respect to reliability and correctness. Therefore it was decided to use the unsatisfactory software concept described above, despite its inadequacies, because nothing else seemed to be available at the time.

Studies are under way, however, to investigate the feasibility of advanced software concepts applicable to digital computerised guidance and control systems. Several possible basic approaches have been considered. One was the use of special high-level languages (HOL), but investigations have indicated that it would, for the moment at least, be difficult to design a HOL suitable for the full spectrum of requirements in the application of digital guidance and control.

A second, universal, method is the application of general purpose high-level languages which translate the algorithmic relations into relatively efficient code. The real-time features are usually implemented by calling up operating system functions via subroutines. Impressive achievements have been realized using such general-purpose programming languages for the generation of real-time programs, one of the outstanding examples being the UK language CORAL 66. The absence of specific real-time features in such languages still remains a problem, which can be overcome by inserting assembly code in order to facilitate the communication between hardware and the operating system. The user is, however, still left with the problems of activating independent parallel tasks, or producing driver routines for the process peripherals.

It is highly desirable to overcome these problems, and the feasibility of using true real-time general-purpose languages is presently being investigated within the framework of the HSF program. During later stages, an attempt will be made to implement such a real-time language in order to investigate its operational aspects. PEARL, a real-time language presently under development by the German Ministry of Research, has been selected as the most advanced candidate for this program. PEARL, which is described in Annex J, has features to describe the algorithmic and the real-time relations of a problem in a high level language. Thus the user is able to describe the interaction of a guidance and control system with its environment (the aircraft, the pilot, the mission, and so on) using precisely defined syntax and semantics. The results of this research could prove most significant; however, much work still remains in order to determine the applicability of this approach to guidance and control systems incorporating hierarchical levels and multi-processor functions. The efficiency and the correctness of the software written in PEARL have to be demonstrated. Compilers and other software tools adequate for the use of the guidance and control engineer still have to be developed.

6. BRITISH EXPERIENCE

6.1 Data Transmission

There has been no UK activity, similar to that reported from other countries, on the development of a standard multiplex bus data transmission specification. UK work has concentrated on the development of techniques and on the exploration of the future data requirements in military aircraft.

In terms of future requirements, contracts are currently being undertaken by the two major UK airframe manufacturers, Hawker Siddeley Aviation (HSA) and the British Aircraft Corporation (BAC), to provide rationalisation of the avionic data transfer between subsystems in military aircraft. HSA is examining the large military aircraft requirements, and BAC, the small strike aircraft.

The work by both firms includes:

- the derivation of data flow models of relevant aircraft types, over both medium and longer time scales;
- the use of these models to classify data into categories;
- the study of possible digital data formats, including present and proposed standards, and the assessment of their suitability for the data categories identified;
- the study of hardware techniques for implementing the formats;
- the recommendation of standards for data transmission in military aircraft, both medium- and long-term.

The UK is particularly interested in the US MIL-STD-1553A, and it is a requirement of the above work that this be included in the formats to be studied. Various UK companies are already involved in work which makes use of MIL-STD-1553, and the Royal Aircraft Establishment (RAE) intends to build laboratory equipment and to examine its usefulness in future trials aircraft. Work at RAE is also involved in electrical multiplexing using the standard.

On the subject of techniques, RAE have been responsible for the technical monitoring of work by Marconi Elliott Avionic Systems Ltd. (MEA) on optical data transmission systems, up to the stage of the preparation of specifications and the supply of a fibre optic link, including cables, transmitter, receiver, connectors, etc. This stage of the work is complete, and the final report includes details of a survey of user requirements within the avionics industry, together with:

- an analysis of these results to arrive at the system specification;
- the results of a parallel survey into technology and components, from which the basic techniques were selected;
- a presentation of the design techniques used in the development of optical transmitter and receiver modules;
- a description of the mechanical features and requirements of an optical data transmission system, together with possible solutions;
- an outline of further work to be done.

This research is now being followed up in two further studies. The first is a searching environmental test programme, being undertaken by the Royal Signals and Radar Establishment (RSRE) in conjunction with MEA, and the second is a study of installation problems in aircraft, undertaken jointly by MEA and BAC.

Research into multiplex data networks employing fibre optic transmission is being pursued, under Ministry of Defence contract, by Standard Telecommunication Laboratories (STL). This work is concentrating on the application of single fibre cables, unlike the MEA development which uses multi-fibre bundles, and so the development of special components is included in the programme, as is the development of terminal equipment which can cope with a wide range of signal levels.

6.2 High Level Language Standardization

CORAL, a Computer On-line Real-time Applications Language, was originally devised at the Royal Radar Establishment (RRE) in 1966. The 'Official Definition of CORAL 66' (see Annex K) was first published by Her Majesty's Stationary Office in 1970 and a second impression was produced in 1973.

CORAL 66 was chosen by the Ministry of Defence as an inter-service standard for military programming, and following its widespread use in military and in civil projects it has been adopted as a standard for real-time and process control systems by the Department of Industry and other Government and Industry bodies.

The language is not intended to compete with established languages in use on large scientific or business machines. It is aimed at implementers of systems with small, even dedicated, computers, where hitherto the use of high level languages has by no means been universal. The objects of its design are as follows:

The compiler must be small enough to run in the production machine or its standby system.

It must produce efficient code, comparable in size to that which would have been produced by an assembler.

It must allow full use of machine-specific hardware and any other special features, but at the same time be capable of implementation on a wide range of machines.

It must be cheap and quick to implement on new machines, and this process should in part be capable of being implemented automatically using such techniques as syntax analyser generators.

The language is designed so that the time taken to compile, load and execute is as short as is consistent with running the compiler in a moderate amount of core with no backing store. No more than one input and output paper tape channel need be used, although, of course, the more facilities there are available in the form of peripheral devices and program development software, the less cumbersome will the process become. Conceptually, all compilers can be single-pass, though in practice there can be a trade-off between the number of passes and the memory required. Typically on a machine with a 4 microsecond add time, the time to compile 3000 instructions of object code will be less than three minutes in a system entirely dependent on paper tape. A CORAL 66 compiler has been produced to run in a machine with only 4K of 12-bit words, using only three passes.

A number of organisations have produced encouraging assessments of CORAL 66. For example, the UK Government Central Computer Agency, in their Report No.4103, reproduced in Annex L indicate that programmer productivity can increase from 4 to 15 times compared with assembler programming, with the consequent code being on average only 30% more. Work at RAE has shown that this excess can be reduced to between 10 and 15% with little effort, and can be eliminated entirely by using high-level language-oriented system design. Other advantages are that the language can be learned quickly, programs and programmers are more transportable and there is more programmer-motivation in its use since it provides more general experience than machine-orientated code.

CORAL 66 compares favourably with other real-time languages, such as PROCOL, PEARL and RTL/2, in terms of its stated objectives, and its uses may best be demonstrated by reference to some of the existing applications. These are:

- Airborne computer systems,
- Shipborne computer systems,
- Battlefield command/control,
- Air traffic control,
- Radar control,
- Message switching systems,
- Control of steel mills,
- Radio station monitoring,
- Control of grid systems,
- Turbine and boiler control,
- On-line heat-exchanger monitoring,
- Graphics support package,
- Data reduction systems,
- Fire control calculations,
- Hospital records system,
- Information retrieval,
- Assemblers,
- CORAL 66 compilers,
- FORTTRAN compilers,
- Simulation language development,
- Operating systems,
- Executives
- Program development systems and support software,
- Utility routines,
- Text editing,
- Disc utilities.

A total of 43 CORAL 66 compilers have been produced or are under development, and training course material, and the courses themselves, have been instituted by the Royal Military College of Science and by the National Computer Centre. The latter organisation also publishes a regular CORAL 66 Bulletin.

Procedures are under way to publish the language as an official Defence Standard, and submissions are being made to the British Standards Institute and to the International Standards Organisation. The Computing Standards Section at RSRE acts as a focal point to provide advice and assistance to users or potential users of the language.

The most important facts, however, are that CORAL 66 is a Government standard language and is independent, for its control and definition, of any single commercial organisation. It is a true standard and has remained unchanged for 5 years. Moreover, there are no plans to change the standard in the future. CORAL 66 compilers are tested by the Ministry of Defence before release, and only approved if they conform to the standard, and CORAL 66 is supported, at little or no cost to the user, in the areas of information services, software libraries, training facilities and courses and compiler development facilities.

7. CONCLUSIONS AND RECOMMENDATIONS

7.1 Concluding Remarks

In accordance with its terms of reference, GCP Working Group 2 has concentrated its studies only in the areas of digital data transmission and high level programming languages. From the information collected, it is clear that similar work in both of these fields is being pursued in all the countries represented on the Working Group, with divergences mainly at the detailed level.

For example, the Group has received details of work on multiplex bus systems from the USA (Annex B), France (Annexes C and D) and Germany (Annex E).

Similarly, each country has developed real time higher order languages, all designed for the same type of task, and all difficult to compare absolutely in any quantitative assessment. Those considered in this report are JOVIAL J73/1 (Annex G) and J3B (Annex H) from USA, LTR (Annex I) from France, PEARL (Annex J) from Germany and CORAL 66 (Annexes K and L) from UK.

The group has not considered any relevant current activities outside the military field, although such consideration may be advantageous in any follow-up activity. For example, an international committee has been considering the design of a long-term procedural language (LTPL) for industrial applications, and this has advanced to the stage where a project plan exists for the implementation of the language, given sufficient funding. The existence of such a language could be significant in terms of its military application potential.

The principal lesson learned from the current Working Group activity is that 4 or 5 people with heavy commitments, meeting twice a year for one or two days cannot provide the necessary resources for the standardization study task. The lack of resources, and the magnitude of the task, have limited the efforts of the Group of the collection and dissemination of information, and have prevented any in-depth study of methodology for the relative assessment of the information. Such assessments as have been done are purely comparisons of the relevant features of the multiplex bus system (Annex F) and the languages (Annex M), and no attempt is made to provide a quantitative assessment, or to indicate how a NATO standard may arise out of this work.

It has become clear, however, that the implications of any study of standardization activities are more far-reaching than just data transmission and languages, and that such a study cannot be applied to guidance and control systems in isolation from the total aircraft systems, of which they are only a part.

It is equally clear that no international standardization activity can force the universal adoption of particular equipment or packaging, or interfere with the methods of procurement extant within individual countries. What is important is the standardization of methodology, such as interface rules and protocols, which allows freedom in the method of satisfying such rules.

7.2 Recommendations

The recommendations of this Working Group, then, are that, since the study of standardization is considered to be applicable across a wider framework than guidance and control, encompassing aspects which are the responsibility of other AGARD panels, such as the Avionics Panel, a Joint Working Group to study the problem of compatibility in digital avionics as a whole should be set up, to carry the work further towards the realisation of the standardization activity.

The principal difficulty in establishing the terms of reference for such a group will be in bounding the problem. Some sort of partitioning into areas for the consideration of specialist sub-groups may be advantageous, and possible areas are considered to be:—

- (a) System architecture,
- (b) Software,
- (c) Multiplexing and interfaces,
- (d) Man/machine interfaces,
- (e) Processor architecture.

It is important that when particular areas of the total task are considered to be ready for definite standardization activity, such areas must be brought to the attention of the NATO Standardization Organisation.

The Group considers that one such area exists now, and this is the standardization of multiplex bus data transmission methodology. The coincidence of current programs in this field in several NATO countries and the technological feasibility of its implementations, as evidenced by the current work, indicate that the time is right for such activity. It is, therefore, strongly recommended that relevant standardization procedures be instituted as soon as possible.

ANNEX A

THE MULTIPLEX DATA BUS

CONTENTS

	Page
1. INTRODUCTION TO MULTIPLEX DATA BUS TECHNOLOGY	A-3
2. BUS ARCHITECTURE	A-3
2.1 Configuration	A-3
2.2 Message Routing	A-3
2.3 Channelling	A-4
3. TRANSMISSION METHODS	A-5
3.1 Multiplex and Modulation Techniques	A-5
3.2 Manchester Bi-Phase Code	A-5
3.3 Litton Code	A-5
4. WORD AND MESSAGE FORMATTING	A-6
5. BUS OPERATION AND BUS CONTROL	A-6
6. REMOTE TERMINAL INTERFACE	A-6
7. TRANSMISSION MEDIA	A-6
8. TRANSMISSION LINE CONSIDERATIONS	A-7
8.1 Lossless Bus	A-7
8.2 Lossy Bus	A-7
9. BUS COUPLING	A-7
10. SIGNAL DETECTION	A-9

1. INTRODUCTION TO MULTIPLEX DATA BUS TECHNOLOGY

As the guidance and control systems of today's aircraft and aerospace vehicles have become increasingly large and complex, ways have been sought to simplify the method of interconnecting the many sensing, control, data processing and other circuits located throughout the vehicles. A point has been reached where conventional wire bundles are too heavy and cumbersome to meet appropriate demands, and the data bus appears to offer a solution to the problem. Here, all the required information can be multiplexed onto a single carrying medium, such as a twisted pair cable, a coaxial cable or an optical fibre, and routed to the data users and sources.

The increase in popularity and the various requirements for the application of data bus systems has brought in its wake the need for extensive studies and laboratory investigations on the many disciplines and techniques that are part of data bus technology. In several NATO countries, the outcome of technological investigations and of the analysis of the requirements on board the aircraft have resulted in firm or preliminary specifications for multiplex serial data busses. These specifications, mentioned in Section 2, are included as Annexes B to E.

- (1) The firm standards set forth by the US Air Force have resulted in MIL-STD-1553A (Annex B).
- (2) In France, considerations of a Comité Technique Integration have led to preliminary specifications laid down in the document "Recommandations provisoires d'un système de transmissions multiplexées sur ligne bus à bord d'aéronefs" (Annex C). A firm specification: "Definition du digibus pour l'échange des informations numériques dans un avion de combat", has been derived by Avions Marcel Dassault - Bréguet Aviation (Annex D).
- (3) In Germany, a technical committee has laid down preliminary specifications "Einheitliche Grundstruktur und Schnittstellen (funktionell, elektrisch, mechanisch) in Flugführungssystemen mit digitaler Signalverarbeitung" (Annex E).

So far, the transmission specifications have been established on a national application basis. Clearly, this has led, as can be seen from the specifications in Annexes B to E, to a proliferation of data transmission techniques. The outcome of the study, however, shows that the basic requirements of the different countries (e.g. the various modes of data exchange or the specified transmission rates) seem to be very similar.

Data busses comprise complex technologies; therefore, it is difficult to compare and assess different concepts and different approaches without knowledge of technical details. In spite of these difficulties, an attempt was made by the Working Group to compare and assess the different bus concepts. The assessment and comparison is mainly based on the different data transmission methods and on the different data organisations utilised within the individual bus concepts. The results are given in Annex F.

Specifications are an outcome of requirements and available technology. It was believed that, in general, the reader of the study report would not be familiar in detail with the technical problem areas; therefore, a short introduction into the general problem areas of bus technology is given here. It should provide the reader with the necessary tools for a better understanding of the bus comparison and assessment and the technology behind the bus specifications from the different NATO countries.

2. BUS ARCHITECTURE

2.1 Configuration

The highway, or serial data bus, architecture is appropriate to systems where data sources and users are distributed over considerable distances. The data exchange between sources and users is routed through a multiplexed digital data bus, and the proper bus operation is controlled and supervised by a bus controller, or bus master. Each source and user connected directly to the bus is equipped with a remote terminal, which contains the electronics necessary to interface the subsystem and the bus. A subsystem can itself comprise several data users and sources grouped in stars around a central remote terminal unit which is connected to the bus. The remote terminal unit provides for central multiplexing and demultiplexing of a subsystem and, therefore, for significant hardware savings with individual small data users and sources, whilst retaining the simplicity and ease of installation of remote terminal units to the data bus.

2.2 Message Routing

Message routing concerns the route by which messages travel from one terminal to another. The majority of applications require the following message routing for data streams:

- (i) direct bus controller (usually a central processor) to terminal transfer, i.e.:
 - commands (individual and global) from a central bus controller to remote terminals,
 - data transfer from a central bus controller to remote terminals;
- (ii) direct terminal to bus controller transfer, i.e.:
 - data transfer from remote terminals to a central bus controller, either program- or interrupt-controlled;

(iii) terminal to terminal transfer, i.e.:

- direct or indirect data transfer from one remote terminal to another, either program- or interrupt-controlled. Indirect transfer means that the data is transferred from a terminal to a bus controller and then on to the receiving terminal.

These data streams have to be carried on the data bus, and they are interlaced or "multiplexed". A remote terminal address must always accompany a sequence of commands, and a remote terminal address plus a user address must accompany a sequence of data.

2.3 Channelling

With regard to the hardware routing of the data streams, several options are available, but only two principal alternatives are nowadays considered. The classification of these alternatives is made on the basis of the number of lines used for carrying the different data streams, and they are, therefore, known as the one-line bus and the two-line bus.

Two configurations of a one-line bus are of interest. The one-line bus system of Figure 1(a) employs the half-duplex method, where supervisory information (commands) and data are transferred on the same cable. This bus architecture requires bi-directional coupling on to the bus cable for each remote terminal and for the bus controller. Half duplex operation is generally preferred to full duplex because of the hardware simplicity of a single line and, therefore, it is one of the most commonly used bus configurations. It is specified by both the US and German requirements (see Annexes B and E). The overall French recommendations of Annex C leave flexibility in selecting either a one- or a two-line bus; however, within the "Definition du digibus..." of Annex D, a two-line bus has been specified.

The one-line bus architecture of Figure 1(b), called a ring bus, requires a different coupler for the input (receive) and the output (transmit) directions, for each remote terminal and for the bus controller. Part of the receiving and transmitting circuitry must be incorporated into the bus line for any module to be connected to a one-line ring bus. Reliability and integrity requirements may, therefore, impose problems, and special technologies may have to be applied to overcome them. One-line ring busses have been proposed for laboratory application but never for aircraft or space vehicles. An example of a one-line ring bus application is the serial CAMAC bus, and for more detail the user can refer to the CAMAC literature.

In a one-line bus configuration, address, command, data and status words for response must be routed through a single bus line. Studies have shown that asynchronous command/response control is superior to other methods for control of such a data bus. In the Figure 1(a) configuration, a message transfer containing a command and a response occupies the data bus for the time required to transfer the serial pulse trains for command and response, plus the delay times required to route the command to the remote terminal and the response back from the remote terminal to the bus controller. These delay times, on a bus of 200 m length using twisted pair cables, are of the order of 1 μ sec and, therefore, cause no serious problems when considering transmission rates around 1 Mbit/sec.

In the ring bus configuration the information (command or response) is always routed unidirectionally; therefore, delay times will not slow down the possible rate of information because of the pipelining effect.

In the two-line configuration of Figure 2(a), one bus line is used for commands from the bus controller to the remote terminals, and the other bus line is used for data exchange and transmission of the status word for response. The bus configuration of Figure 2(a) implies unidirectional coupling on the command bus to the remote terminals, and bi-directional coupling on the data bus from and to the remote terminals. Compared to the one-line bus, the amount of circuitry for coupling is higher. As two lines are used for the bus, message transfer can be organised so that the delay times do not slow down the maximum permissible rate of information determined by the clock and data code. This architecture is the one specified in the French system of Annex D, as mentioned above.

The two-line bus configuration of Figure 2(b) uses one line for each transfer direction so that unidirectional data transfer is achieved on the bus. Unidirectional coupling is required for each remote terminal; the command bus transmits and the response bus receives. The use of two bus lines also allows for the organisation of the message transfer so that the delay times do not slow down the maximum permissible information rate as determined by the clock and data code. Full duplex systems, however, need more hardware for implementation.

Two-line bus configurations with unidirectional transfer offer additional solutions as ring busses as shown in Figure 2(c). No application for this bus architecture, however, has yet been found in the aerospace field.

All bus systems (one- or two-line bus) allow for a system architecture involving either partial or complete redundancy.

3. TRANSMISSION METHODS

3.1 Multiplex and Modulation Techniques

With regard to the multiplex technique for digital avionic data busses, in aircraft systems one is normally dealing with a large number of low frequency or discrete signals. This means that the time division multiplex (TDM) method, where signals are time-shared on the data bus, is the most suitable technique. The other alternative, frequency division multiplexing (FDM), would be suitable for high frequency and video signals with relatively low channel capacity.

With respect to the modulation (coding) technique, all the bus systems which have been considered by the Working Group require the signals for data exchange to be transferred over the data bus in digital pulse code modulation (PCM) form, wherein each analogue or digital signal is represented by a separate serial train of discrete binary pulses. PCM techniques are preferred for several basic reasons, such as the absence of transmission degradation and the fact that much aircraft data already exists in digital form or must be converted to digital for processing.

Two general classes of modulation are available: carrier modulation and baseband modulation. Carrier modulation techniques involve the modulation of a carrier by the digital data, whereas baseband modulation techniques do not involve a carrier and are simpler to implement. Baseband modulation techniques, therefore, receive prime consideration.

The baseband modulation (or data code) technique used is of great importance. Basic requirements set out by the bus operation leave a limited choice of codes which can be used. These requirements are:

- The data code must allow ac coupling and shall, therefore, not be susceptible to dc level drift.
- As clock lines are not used on the bus, the data code selected must facilitate clock recovery by the remote terminals.
- The data code selected must allow for easy data framing (words or messages).
- Code errors shall be detectable, as far as possible.

Most codes do not meet the requirements to facilitate clock recovery and to have no base line drift. Examples of satisfactory codes are the Manchester bi-phase code, the Litton code, ternary bipolar codes, and variations of these.

Further codes meeting the requirements would be the group codes with selected combinations, and the PSK (phase shift key) codes, which utilise two logic levels and two transition states for the bit combination 00, 01, 10, 11. The complexity of the circuitry to implement these codes, however, is considerably higher than in the aforementioned; therefore, they are not used for modulation in aircraft data bus systems.

3.2 Manchester Bi-Phase Code

Most specifications for bus systems which have been proposed by the different NATO countries call up the use of the Manchester bi-phase code for modulation. As can be seen in Figure 3, a logic 1 is coded as a bipolar 10, i.e. a positive pulse followed by a negative pulse, whilst a logic 0 is a bipolar 01, i.e. a negative pulse followed by a positive one. A transition through zero occurs at the mid point of each bit, be it a 1 or a 0, and these transitions through zero can be used to regenerate the clock waveform.

As far as synchronisation of the serial pulse train is concerned, bit sync can be accomplished by regenerating the clock from the received waveform, as mentioned above. Several possibilities exist for frame synchronisation, which is required to recognise the start of a word or message. The most usual method is the use of an invalid waveform, which can be uniquely recognised in a train of pulses; such a waveform can have, for example, a width of 3 bit times, being positive for the first $1\frac{1}{2}$ bit times and negative for the following $1\frac{1}{2}$ bit times. Now if the first data bit following such a sync signal is a logic 0, ambiguity will result since the end of the sync waveform is not well defined. To overcome this problem, the sync waveform is often followed by a logic 1, increasing the sync pulse length to 4 bit times, but simplifying detection. These synchronisation methods are shown in Figure 4.

It is not normally possible to synchronise on, say, a particular bit pattern because it is difficult to ensure that this pattern will never appear in the data word. Another possible frame synchronisation method is to employ a pause at the zero voltage level. The synchronisation of a message is then achieved by detecting the first bit of the message.

3.3 Litton Code

The Litton code uses antipolar pulse pairs to indicate logic 0's and 1's, where the equidistant pulses are separated by spaces at zero level. A logic 1 is coded as a positive pulse followed by a negative pulse, and a logic 0 is the opposite. The Litton code, and all other ternary codes, require three-level detection (plus, minus and zero).

The clock extraction for bit synchronisation can be most easily achieved by simple full wave rectification of the pulse sequence, the result being a sequence of equispaced pulses. Frame sync can be achieved either by introducing pauses with zero voltage level or by the use of invalid waveforms. As in the case of the Manchester bi-phase code, the

invalid waveform used for synchronisation has a width of three bit times, in this case three positive pulses followed by three negative pulses. It can be seen from Figure 5 that the introduction of an invalid waveform for frame sync implies no degradation of the clock recovery capability which is important for bit sync; this is not true in the case of the Manchester bi-phase code. With respect to clock recovery, the Litton code has distinct advantages and its use has been proposed for bus systems for space applications.

Ternary bipolar codes have similar advantages to the Litton code but have not so far been used for applications of bus systems in the avionics or space field, and, as stated, all ternary codes have the basic disadvantage that three levels must be detectable.

4. WORD AND MESSAGE FORMATTING

Methods for word and message formatting are best assessed and compared in terms of message contents. The information on the data bus, in all the serial bus systems proposed, contains the data pulses, which can also be used for bit sync, and special signal symbols signifying the beginning (and/or end) of words and messages. The messages comprise three types of word: command, data and status. The command word usually contains an address and a function part and has a length around 16–24 bits. The command or supervisory instructions may designate:

- addresses of information sources and sinks (user subsystem designations);
- quantity of data to be transferred (e.g. number of bytes);
- when to begin and stop a data transfer;
- time reference information;
- error control information.

The data word has either fixed or variable length; however, one byte cannot usually be sub-partitioned. The status word comprises address and status information. The message formatting arrangements proposed for the various individual busses are quite different – details on how the words and messages are specified can be seen in Annexes B to E. The conclusion can be drawn that the different bus concepts proposed from the NATO countries are the same in concept but quite different in detail, particularly with respect to the formatting. Since the on-board requirements within the different NATO countries are quite similar, however, it is believed that a common formatting arrangement would be possible if there should be a need for the busses to be standardized within these countries.

5. BUS OPERATION AND BUS CONTROL

Studies have shown that, although many techniques for bus control exist, centralised control is generally the superior method. All bus systems proposed, either for military, space or industrial applications, therefore, require that information transmission on the bus is controlled by a bus controller which initiates all transfers. The bus controller is a critical element in case of failures, and, therefore, some bus proposals provide for dynamic handing over of bus control to alternative remote terminals capable of taking on the role. All proposed systems, however, specify that only one bus controller can be active at any one time, and bus control can only be transferred to alternative stations after completion of a message.

6. REMOTE TERMINAL INTERFACE

The subsystem/remote terminal interface is often achieved by dividing the remote terminal into two units: the multiplex terminal unit, which performs the serial data bus interface, and the subsystem interface unit, which performs the interface to the user subsystem. Such an arrangement is shown in Figure 6. The interface between the multiplex terminal unit and the subsystem interface unit may be either serial or parallel. The subsystem interface unit may be a coupler to a parallel bus of a user subsystem or part of the bus controller of a parallel bus itself. In either case, the interfaces should be standardized. Modularity is desirable because it leads to:

- flexibility in remote terminal configurations;
- possibilities which allow the use of MSI/LSI techniques to be cost-effective.

7. TRANSMISSION MEDIA

The technologies and the transmission medium determine, to a large degree, the error rate and the power requirements of the multiplexed bus system. Some types of medium are:

- (1) Electromagnetic radiation
- (2) Optical links
- (3) Wire cable
 - (a) Coaxial
 - (b) Twisted shielded pair

Electromagnetic radiation, guided by means of dielectric wave guides has received considerable attention in some areas because of the possibilities of high bandwidth and high data rates. Wave guides, however, are bulky and susceptible to physical damage, and are not seriously considered for application in aircraft.

Optical data transmission shows great promise for the future. Using fibre optics, bandwidths of the order of 10^{11} Hz can be achieved. Coupling and termination of fibre optics are problems at present being solved, and, since typical aircraft data transmission requirements can be satisfied, generally, with data rates of the order of 1 Mbit/sec, there has been no great spur for optical applications up to now. The increasing use of non-metallic composite materials in modern-day aircraft, however, may speed up the adoption of fibre optic data transmission.

Wire cables have been the principal media for multiplex transmission up to the present. Coaxial and twisted shielded pair cables have shown the greatest potential. Coax cables are usable to high frequency applications (up to 100 MHz), but the susceptibility to electromagnetic interference (EMI) and radiated noise are higher, due to the fact that the shield carries signal current.

Twisted shielded pair cables are usable to frequency applications up to 10 MHz. The EMI susceptibility and the radiated noise are less, since the signal carrying conductors are shielded, and this is the reason why twisted shielded pair cables are favoured for aircraft and space, at the relatively low data rates associated with current avionic applications.

8. TRANSMISSION LINE CONSIDERATIONS

8.1 Lossless Bus

The lossless bus line is the classical design approach to the transmission line. Figure 7 shows its basic configuration. By terminating the bus line with its characteristic impedance, Z_0 , at both ends, reflections are kept to a minimum, provided that the impedances of the remote terminals are also properly controlled. In the receive mode the remote terminals can be held at a high impedance, which minimizes the effect on the bus line, whilst in the transmission mode the remote terminals must be brought to a low impedance. This bus line configuration is "lossless" only insofar as losses are kept to a minimum and not intentionally introduced. Thus the lossless line needs only low transmitter power to achieve good signal-to-noise characteristics. The advantages of lossless bus lines are: minimisation of reflections, hardware simplicity and relative ease of bus extension. The major disadvantage is its inability to withstand line faults. As can be seen in Figure 7, a line-to-line short at any point places a direct short across the transmitting terminal and the entire system is lost. If an open-circuit occurs on the line, then all remote terminals past the fault are lost, and a severe mismatch results, probably causing reflections which would prevent successful bus operation.

The only method of protection against line faults is to provide sufficient redundant bus lines to ensure adequate mission and system reliability. This form of redundancy, however, requires active switching elements, and, therefore, adds to system complexity.

8.2 Lossy Bus

The lossy bus configuration is shown in Figure 8. The insertion of losses by means of resistors conflict with the classical approach but, nevertheless, offers some advantages. By providing loss resistors at each remote terminal, and by looping the bus lines (not to be confused with a ring bus), a configuration is obtained which is more resistant to line faults. A bus configured in this manner can tolerate one fault, either an open or a short circuit, without the loss of any of the remote terminals. Redundant configurations can be designed without the need for active switch elements, as shown in Figure 9.

This approach, however, does have several disadvantages. The different lengths of the two paths round the loop may result in a signal arriving at a given terminal at different times, and this limits the achievable data rate. In addition, the receivers must be designed to operate on a wider dynamic range, and more signal power is required.

The disadvantages of the lossy line appear to outweigh the advantages, since none of the proposals examined recommend the use of this approach.

9. BUS COUPLING

Several methods are available for coupling the remote terminals to a common bus line. As all proposed bus systems specify either coax cables or twisted shielded pair cables for transmission media, the following coupling methods must be considered.

Direct dc coupling: provides simple and inexpensive means, but is not feasible where autonomous units use separate power supplies. It offers no isolation between remote terminals.

Photo coupling: provides excellent isolation and dc coupling, but is temperature sensitive and requires high-gain amplifiers for signal detection. Little reliable information is available so far. Power consumption is high.

Capacitive ac coupling: provides dc isolation but must be bandpass filtered at high frequencies.

Inductive ac coupling: with transformers, provides dc isolation and good matching capabilities.

Since transformer coupling has been proposed for all military aircraft multiplex bus systems considered in the report, some technical consideration must be given to the problems.

From the transmission standpoint, bus wire lines are four-terminal networks and, as such, have frequency-dependent transfer characteristics. The characteristic impedance of the bus line, i.e. the ratio of the voltage to the current, is equal at any point provided the line is assumed to be infinitely long.

$$Z_0 = \frac{V}{I}$$

Z_0 is real, with no complex part, for ideal transmission lines, and the same applies to finite lines, provided they are terminated with Z_0 .

A transformer coupled bus can be operated in two different modes, the voltage and the current mode. The equivalent circuits in Figure 10 show that a remote terminal which is not in a transmission mode should have a very high impedance in the voltage mode and zero input impedance in the current mode, in order to avoid discontinuities in the transmission lines (bus lines). Since it is easier to implement a short circuit, the voltage mode is preferred for bus applications; besides, the voltage mode has advantages in the case of single point failures. All bus systems proposed by the NATO countries, therefore, specify the voltage mode.

The most simple method of decoupling a remote terminal from the bus in the case of a single point failure is to put a resistor in series with the secondary coil, as shown in Figure 11. In the case of a short circuit in the remote terminal, the resulting load imposed on the cable is R_s . This load will cause a point of discontinuity and will not affect proper bus operation for the remaining remote terminals.

Using simple network analysis in Figure 11, one computes the transmission coefficient in the case of a remote terminal short circuit to be:

$$n = \frac{1}{1 + Z_0/2R_s}$$

and the reflection coefficient to be:

$$r = -\frac{1}{1 + 2R_s/Z_0}$$

From this, it can be seen that the adverse effect of a failure in the case of a short circuit decreases with higher values of R_s . On the other hand, there are disadvantages in the transmission mode, where, in normal operation, the useful signal on the bus is reduced by the voltage divider caused by R_s , by a factor of $-r$. For practical systems this means that remote terminals require relatively high power for transmission, and the use of a decoupling resistor is far from optimum with respect to power consumption.

If resistors are being used for decoupling, additional disadvantages arise with respect to the pulse transmission characteristics. If the effect of the capacitor (shown in the equivalent circuit of Figure 11) is neglected, one concludes that the rise time of the pulse will be increased with R_s . The same is true of the slope.

The use of an additional transformer will give ideal decoupling for short circuit failures. Figure 12 shows the arrangement and the corresponding equivalent circuit.

The current on the bus lines induces two magnetic fields, h_1 and h_2 . The transformer is wound so that these magnetic fields have inverse directions; thus the overall inductance of the transformer is zero. Using Kirchhoff's laws, the equivalent circuit yields:

$$\frac{V_3}{V_0} = \frac{Z_0}{[Z_0 + jwL \cdot (1 + K)] \cdot \left[1 + \frac{(R + jwL)}{(R - jwKL)} \right]}$$

where K is the transformer turns ratio and w , the radian frequency. If $K = -1$, i.e. the transformer is ideal, the ratio becomes:

$$\frac{V_3}{V_0} = 0.5$$

which is independent of R . This means that the transformer circuit imposes no load on the bus, even in the case of a short circuit failure. An additional transformer thus provides excellent protection against single point failures.

If the remote terminal is to transmit, however, the main inductance of the transformer is the load and this blocks the transmitting signal from the bus. An additional winding must, therefore, be provided, as shown in Figure 13, together with the corresponding equivalent circuit. The leakage inductance and the capacity of the equivalent circuit of the transformer are neglected. When the remote terminal is in the transmit mode the switch is closed, and this balances the main inductance, L_m , of the additional transformer.

The advantages of transformer decoupling as a protection against single point failures, then, are:

- freedom from losses;
- no deterioration in the bus system in the case of a short circuit;
- less distortion of the signals.

The disadvantages are that a failure of the additional transformer leads to a failure of the bus, and that stubbing is difficult to achieve. The main advantages of transformer dc coupling are realised in the space application because of the very low power consumption.

10. SIGNAL DETECTION

The selection of the receiver depends on the:

- code,
- formatting,
- error probability,
- tolerable complexity,
- tolerable power consumption.

If frame synchronisation in a bi-phase implementation is performed by using zero level pauses, then the signal-to-noise ratio is decreased by 6 dB. This is because the pause introduces an additional state to be detected - i.e. zero - and the detection level, therefore, has to be $V_0/2$, whereas in the case of continuous data streams (no pauses) the detection level can be set to zero (Fig.14).

Figure 15 shows what is called an optimal receiver. The data is correlated (multiplied) with the clock bit stream and then integrated and compared. This method requires either a special clock line or a "flywheel" circuit for clock detection because synchronous data is needed. The flywheel must have a large time constant, so this method can only be used if the message is preceded by a signal of at least 40 bits after any pause, in order to synchronise the flywheel. The complexity of an optimal receiver, therefore, is considerable.

There are a variety of suboptimal receivers. A simple receiver, as shown in Figure 16, consists of a filter, to remove sidebands of the data code, and a comparator. The clock signal is either generated with a flywheel (synchronous receiver) or extracted continuously from the data bit stream (asynchronous receiver). Flywheels are usable only if continuous data streams are on the bus or if long sync patterns precede a message, as explained above, and, therefore, asynchronous receivers are normally used for bus systems. Figure 17 shows the block diagram corresponding to the arrangement of Figure 16, together with the related waveform timing diagram. Distortions are due to reflections and EMI. Investigations have revealed that the noise caused by EMI is below $20 \text{ mV}_{\text{eff}}$ if the electromagnetic field is below 10 V/M .

The signal power used for transmitting bus signals must be limited by the tolerable radiated noise involved with the transmission. Using twisted pair cable and, for example, MIL-I-6181D specifications, the signal power can be up to around 100 W. Using special shielded coax cables, the transmitting power can be even higher. This indicates that the signal power is only constrained by the tolerable energy dissipation.

In designing a bus system, one can consider using higher signal levels and less optimal receivers with less energy loss in order to achieve comparable signal-to-noise ratios to optimal receivers. In a bus system, there is only one remote terminal at any one time which is in the transmitting mode; all the others are in the receiving mode. If, for example, each receiver of a bus system, with 32 remote terminals, needs 10 mW less energy, then the signal power could be increased by as much as 3 W for the same energy dissipation on the bus system.

The error rates of several different receivers as shown in Figure 18. The lowest bit error probability is achieved with optimal receivers, whilst suboptimal synchronous receivers have higher bit error probabilities. Measurements indicate that for asynchronous receivers to have comparable error characteristics then the signal-to-noise ratio must be some 2-3 dB better.

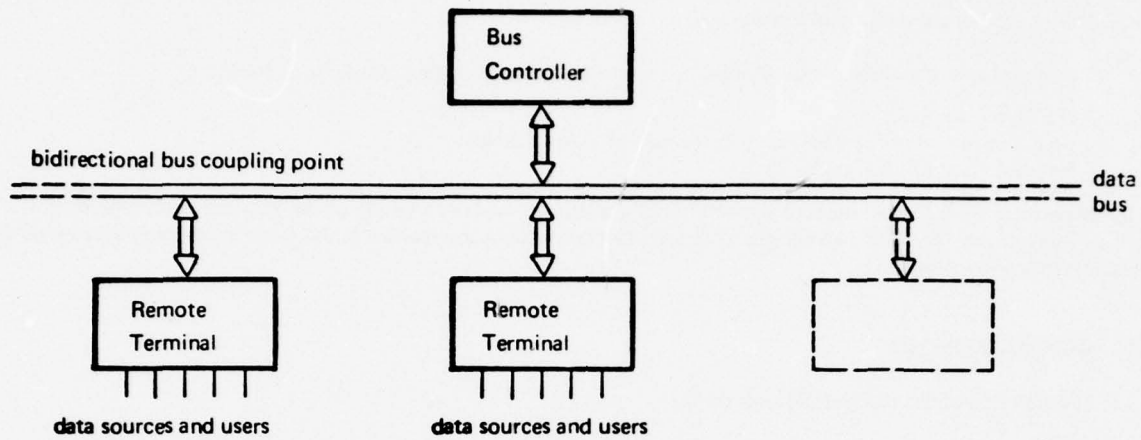


Fig.1(a) One-line bus with bi-directional transfer

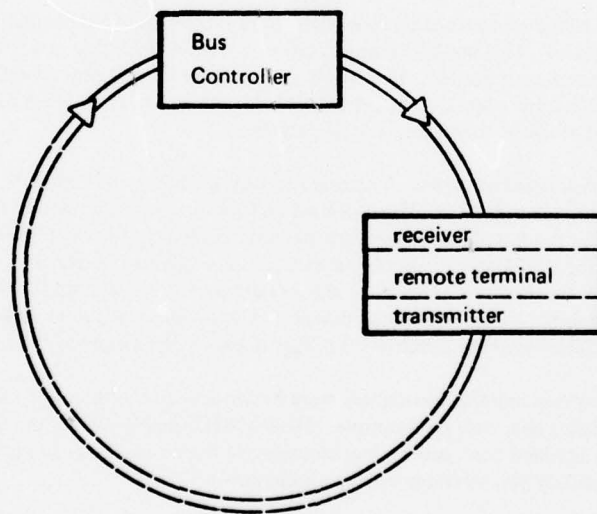


Fig.1(b) One-line ring bus with unidirectional data transfer

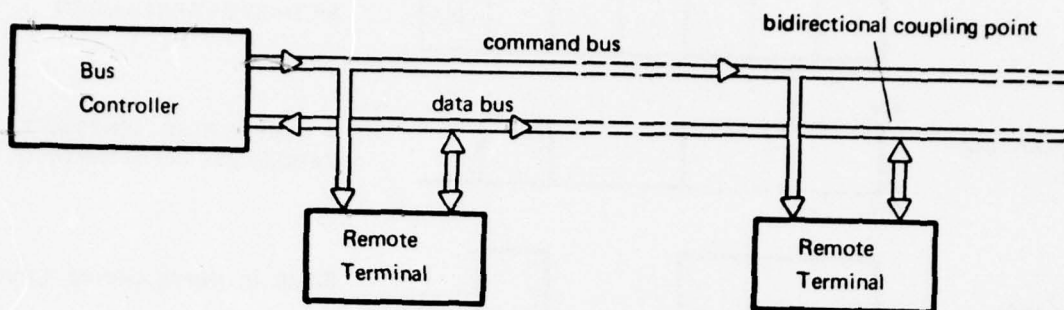


Fig.2(a) Two-line configuration with separate command and data bus

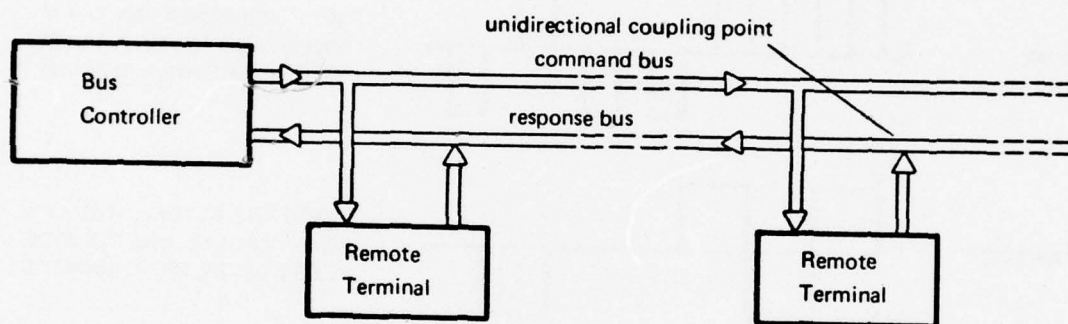


Fig.2(b) Two-line bus configuration with unidirectional data transfer

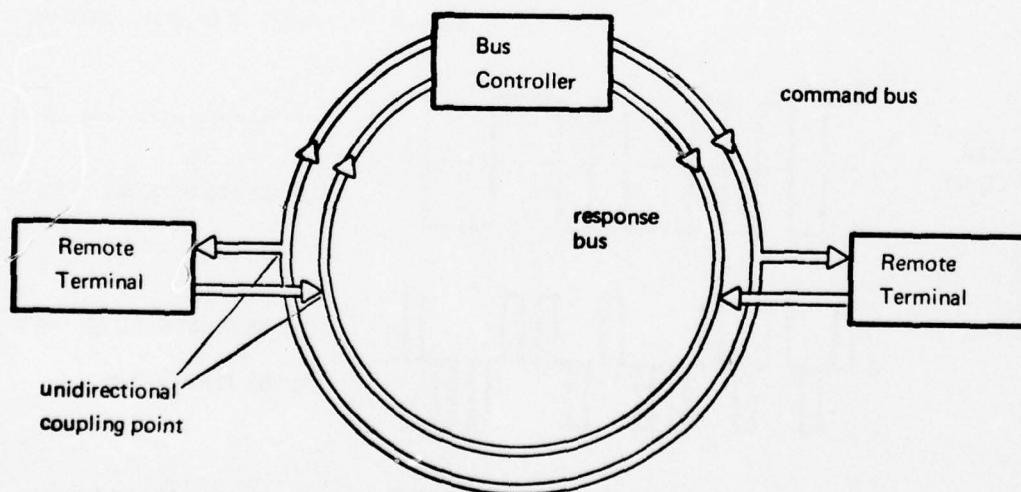


Fig.2(c) Two-line ring bus architecture using unidirectional data transfer

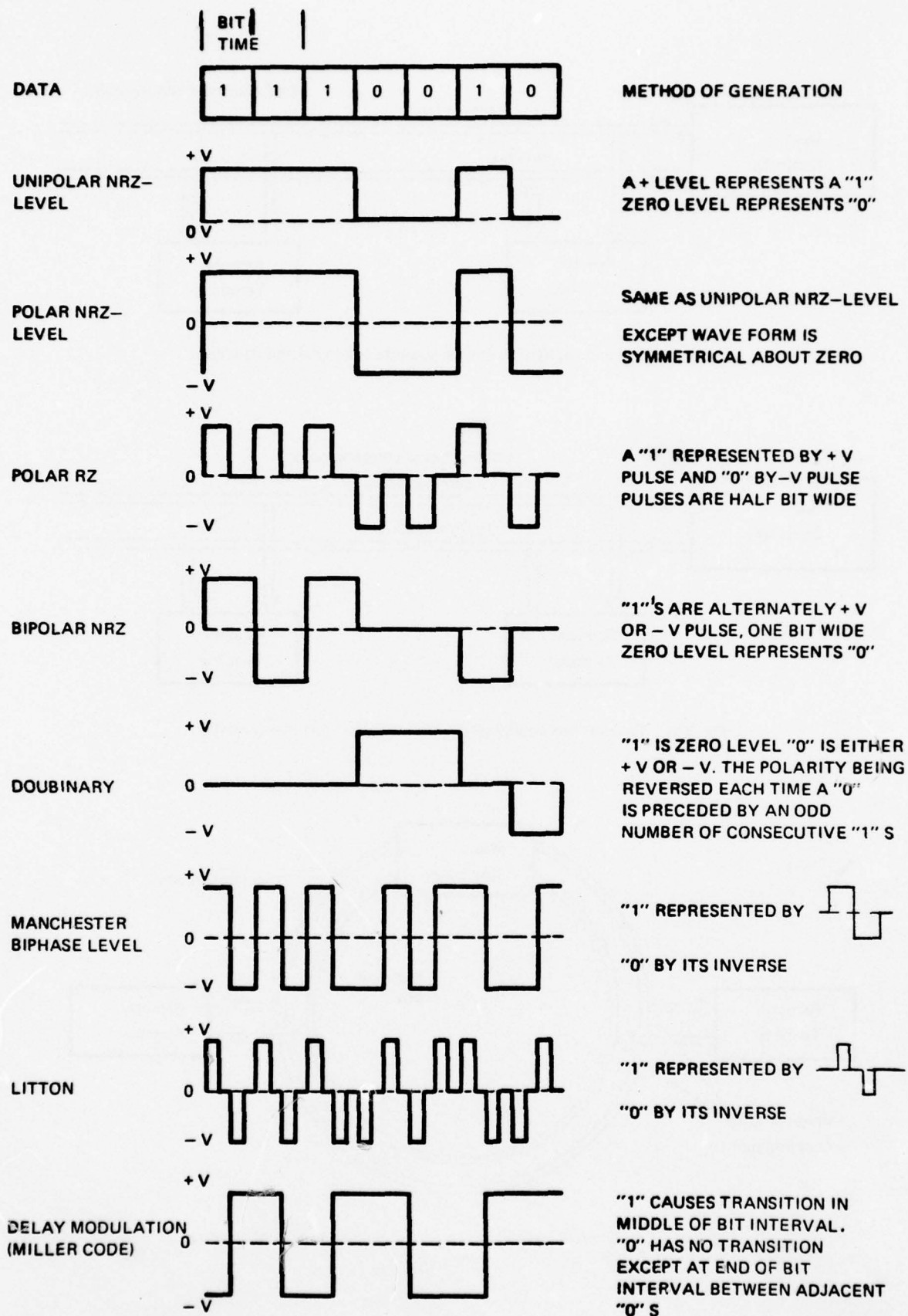
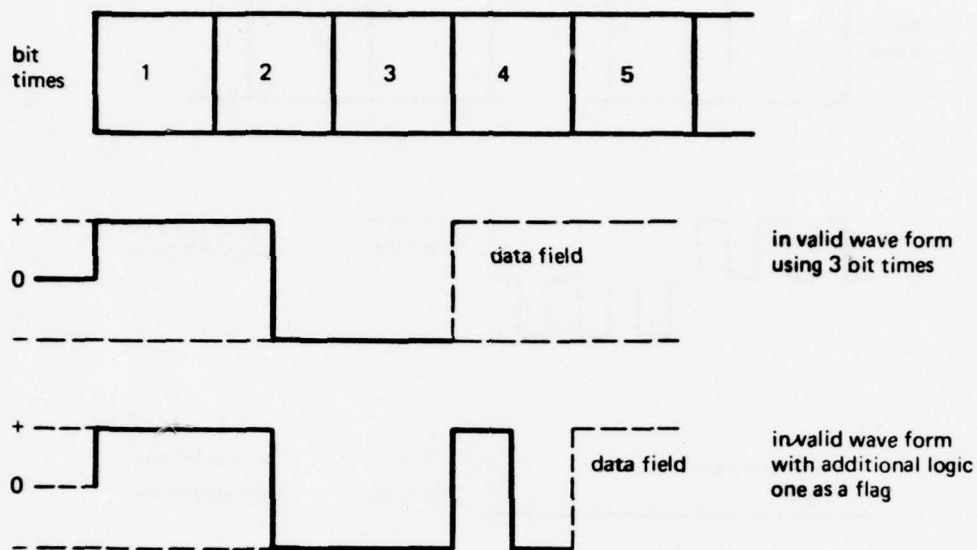
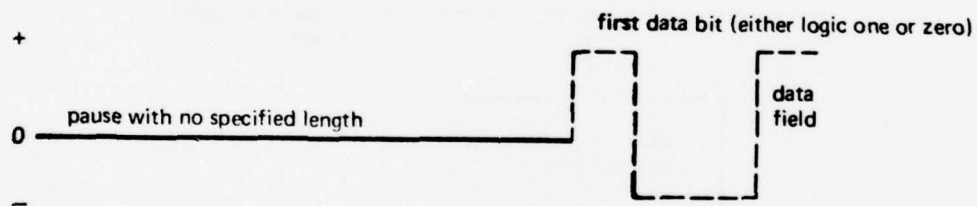


Fig.3 Baseband modulation techniques



Unique wave forms for frame synchronisation for the Manchester Bi-Phase code



Pause with zero voltage level for frame synchronisation

Fig.4 Frame synchronisation for the Manchester bi-phase code

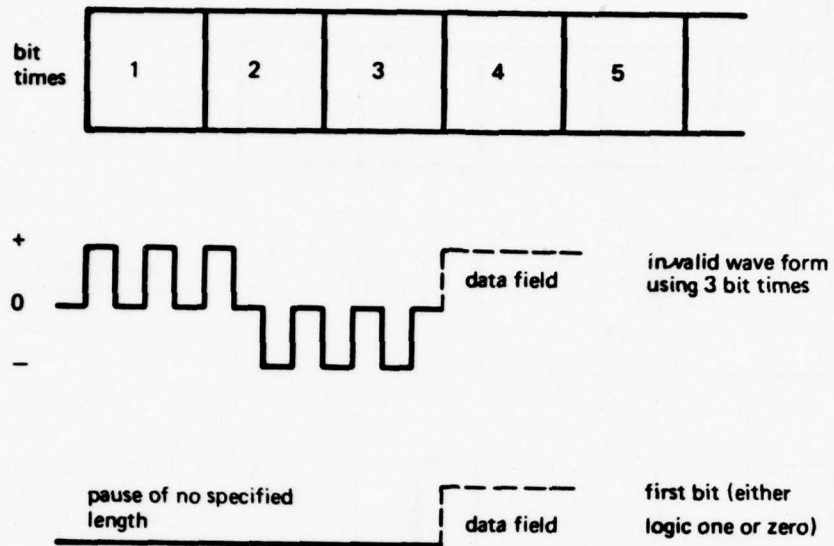


Fig.5 Frame synchronisation for the Litton code

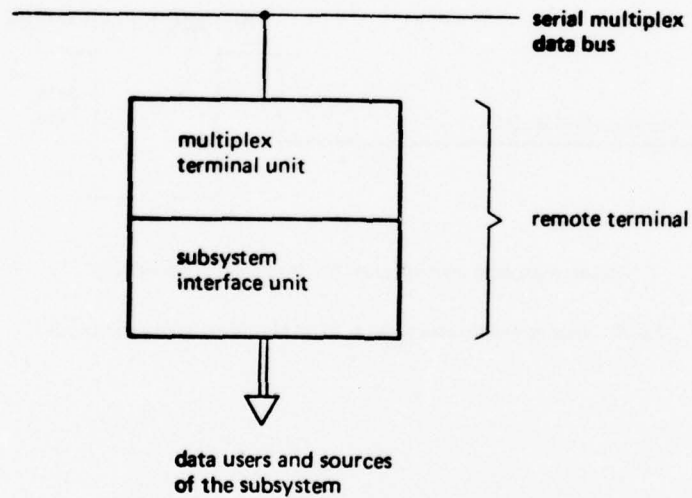


Fig.6 Block diagram of a remote terminal

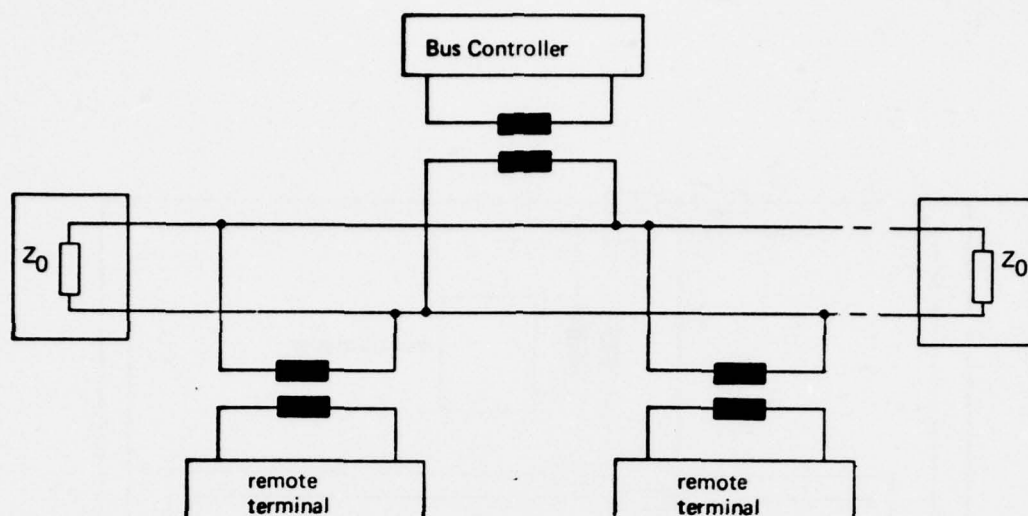


Fig.7 Configuration of the lossless bus line

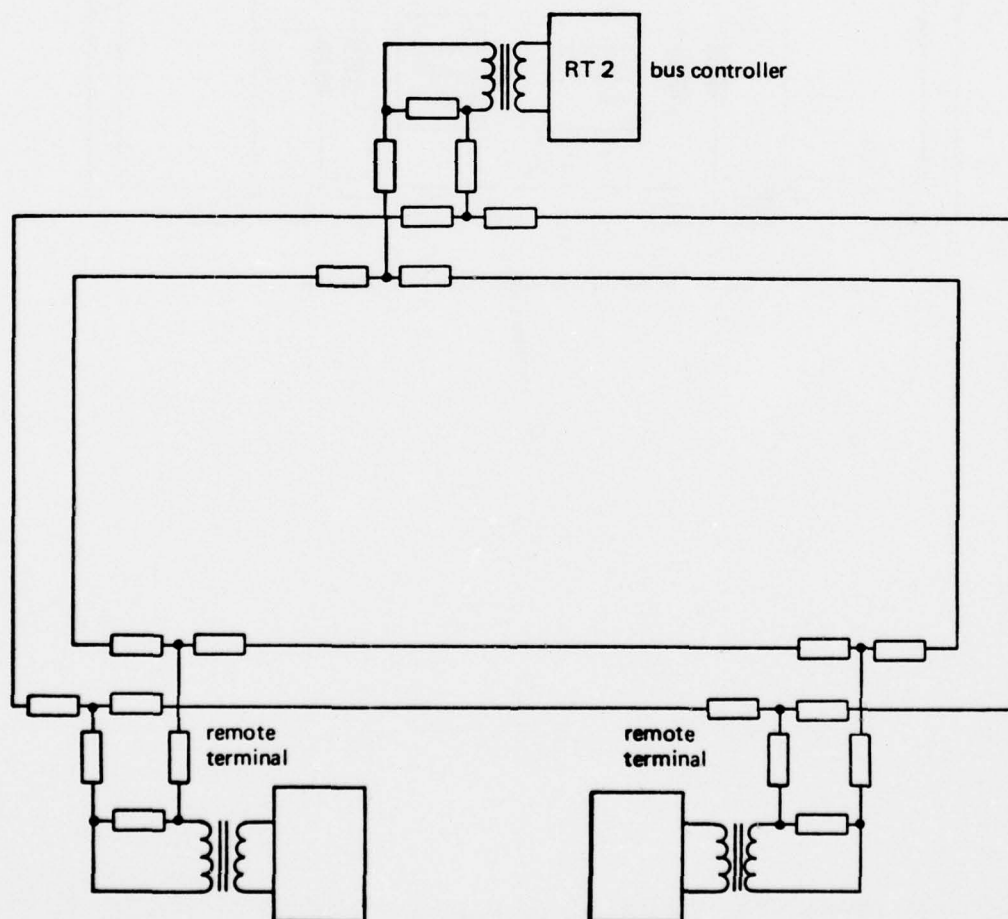


Fig.8 Configuration of a lossy bus

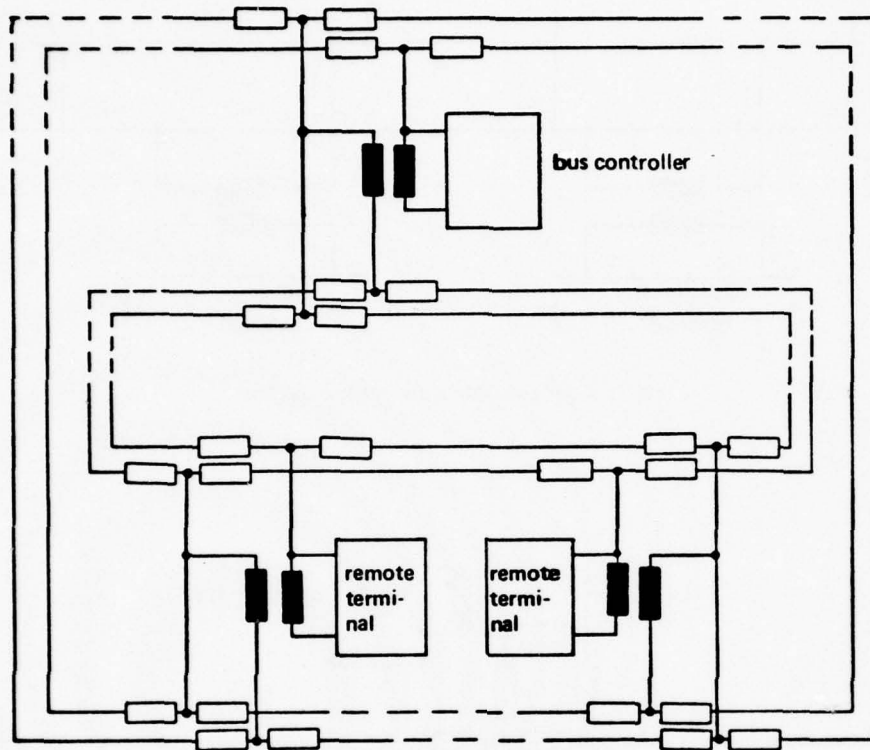


Fig.9 Redundant configuration of a lossy bus

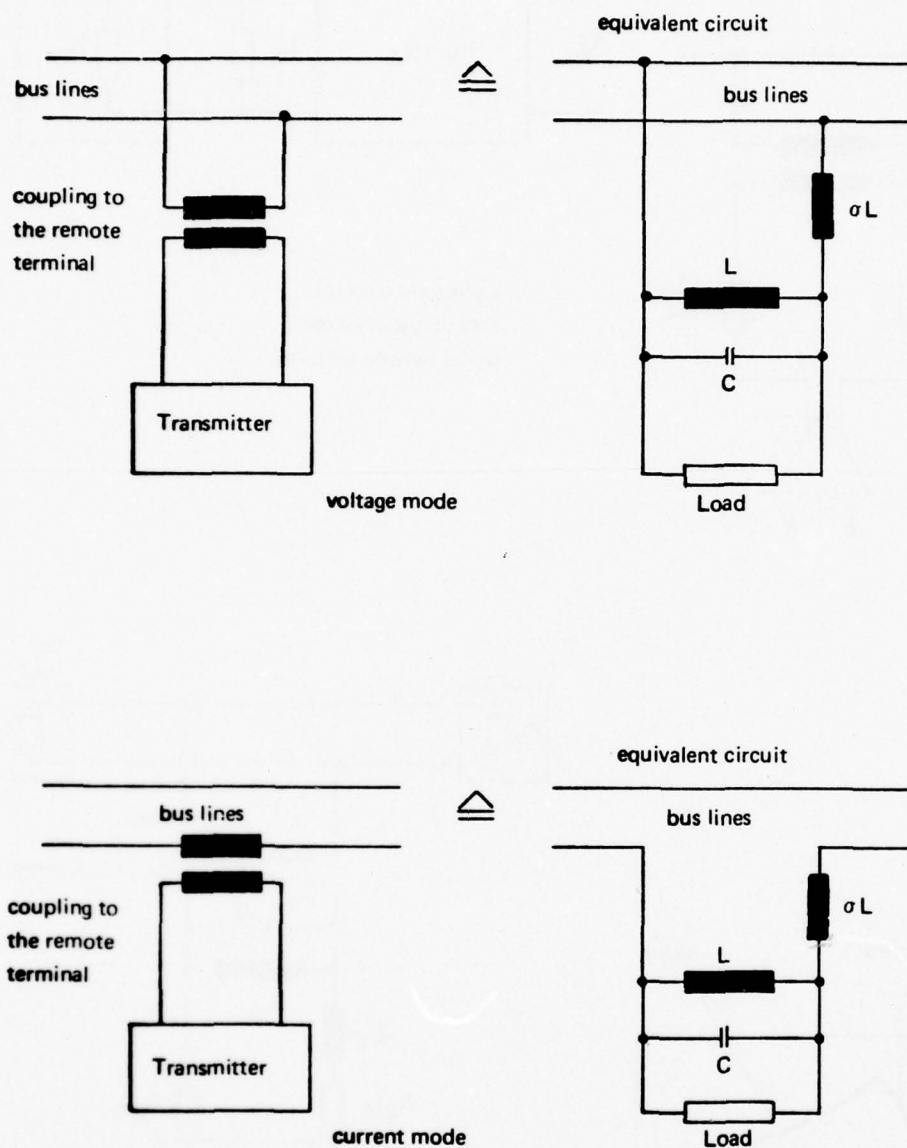


Fig. 10 Voltage and current mode for transformer coupling

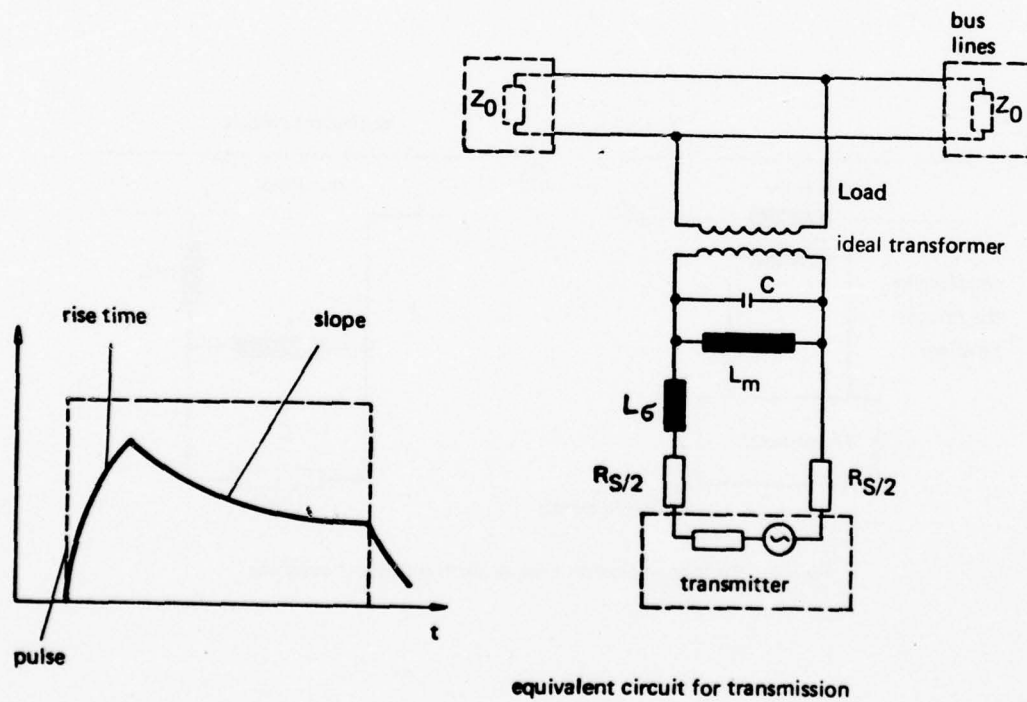
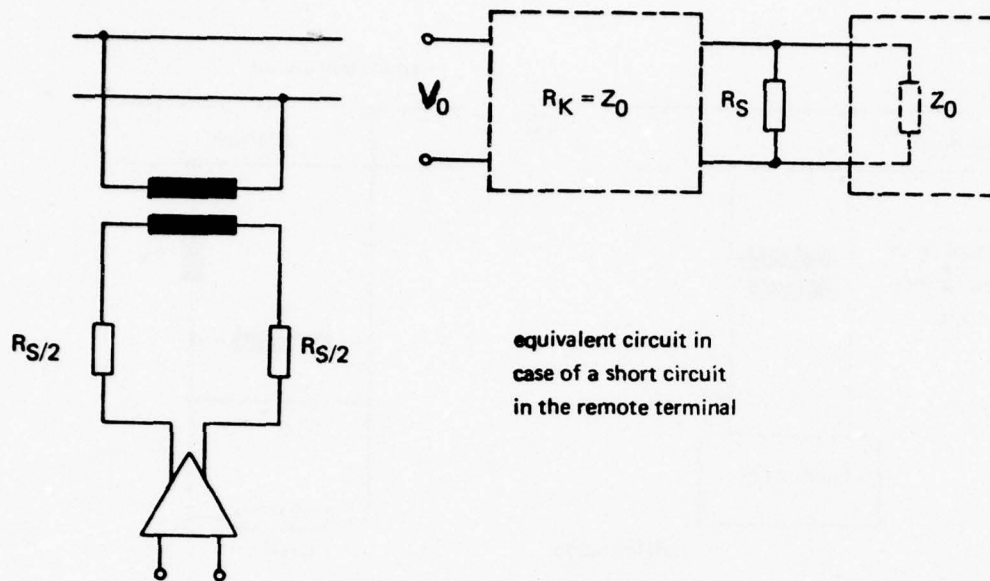


Fig.11 Decoupling with a resistor as protection against single point failures (especially short circuits in the remote terminal)

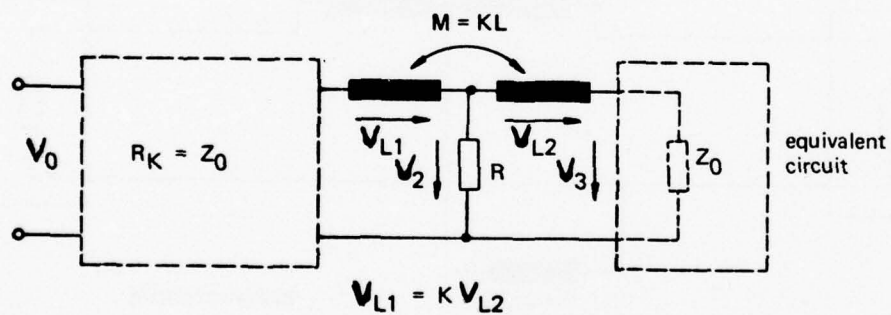
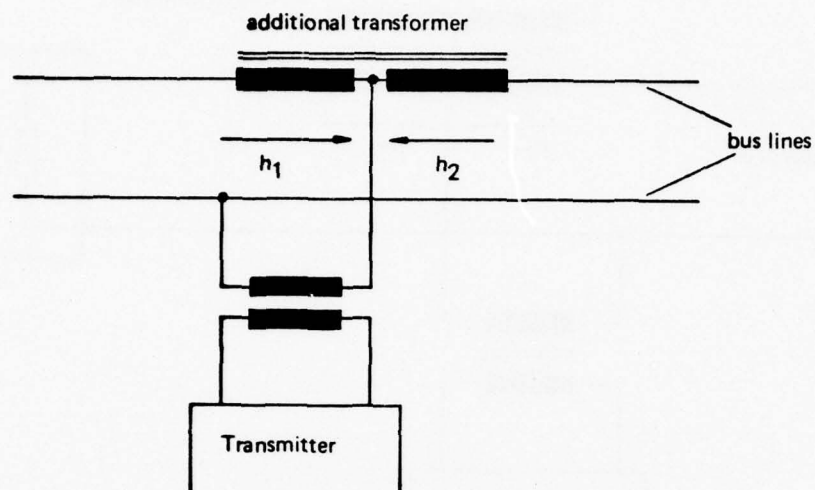


Fig.12 Decoupling with an additional transformer as a protection against single point failures (especially short circuits)

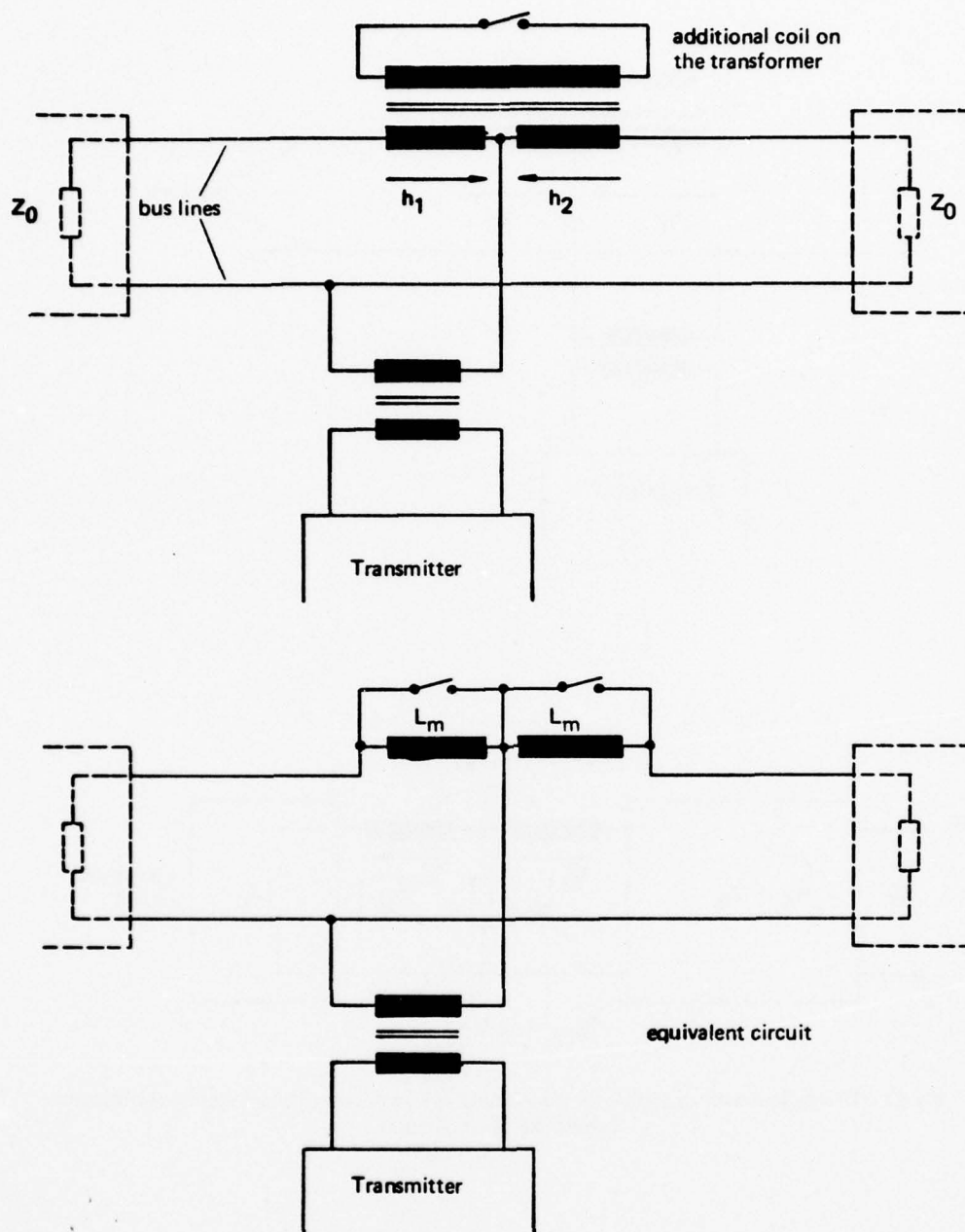


Fig.13 Transmission mode using an additional transformer for decoupling the remote terminal in case of a failure

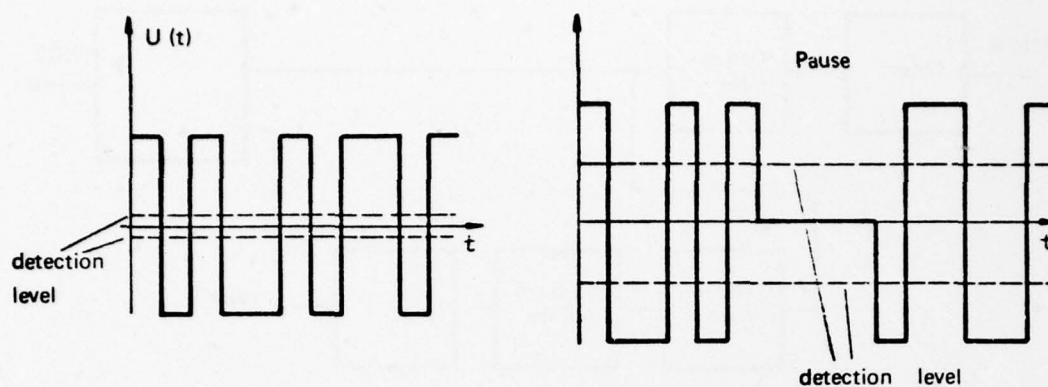


Fig.14 Signal detection level with bi-phase and bi-phase ternary codes

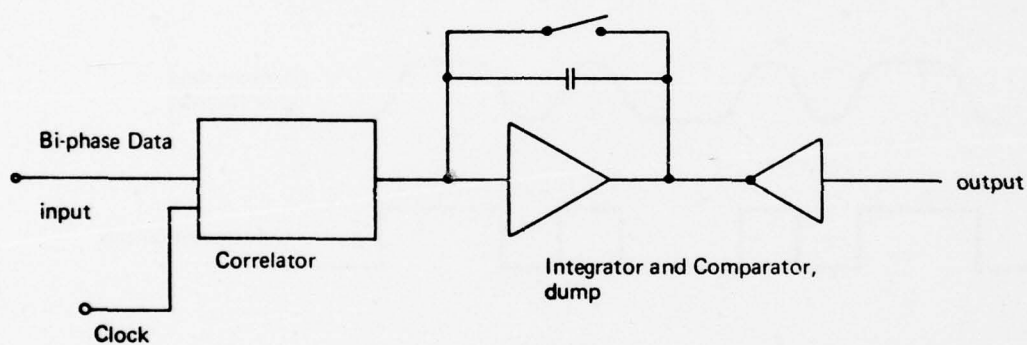


Fig.15 Principle of an optimal receiver

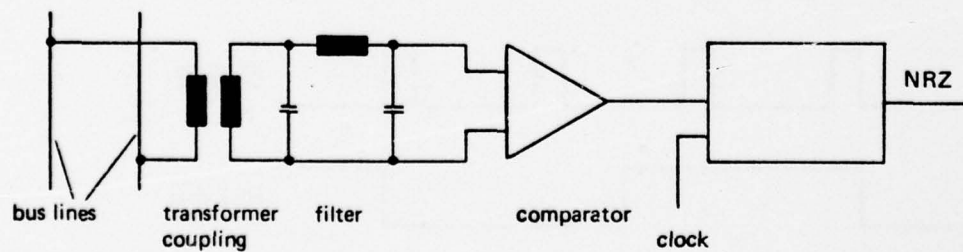
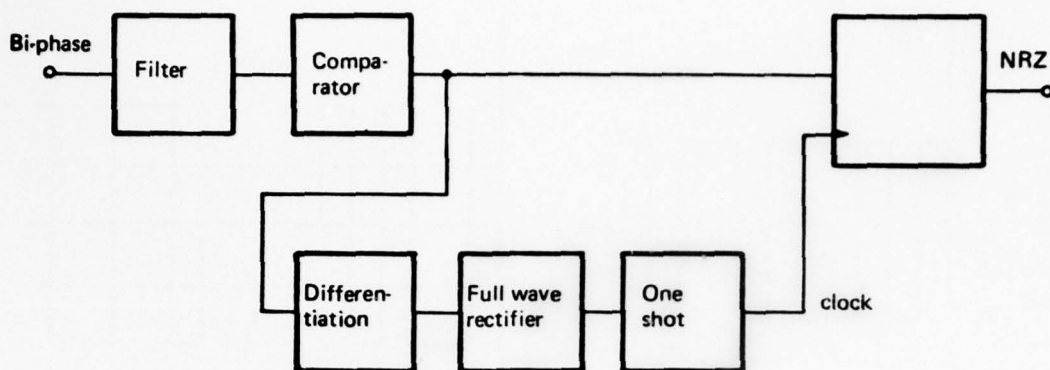


Fig.16 Principle of a suboptimal receiver



Block diagram of the asynchronous receiver

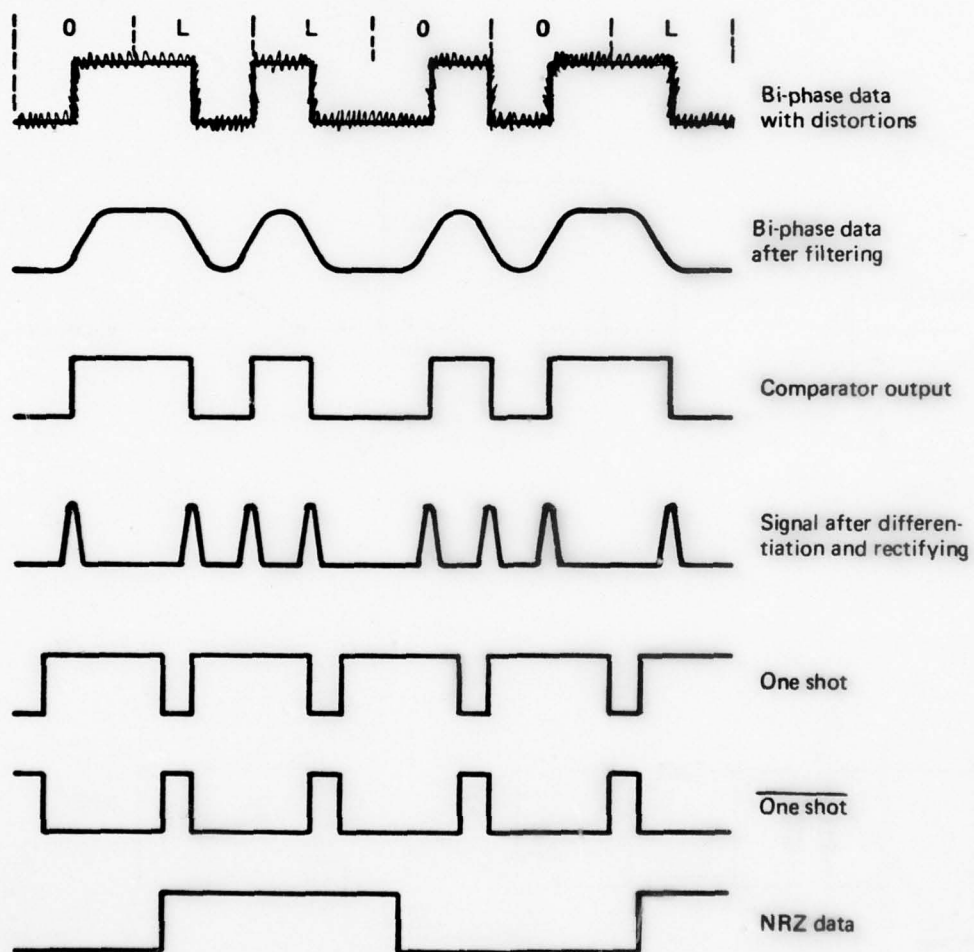


Fig.17 Block diagram and waveform timing diagram of a synchronous bi-phase data receiver

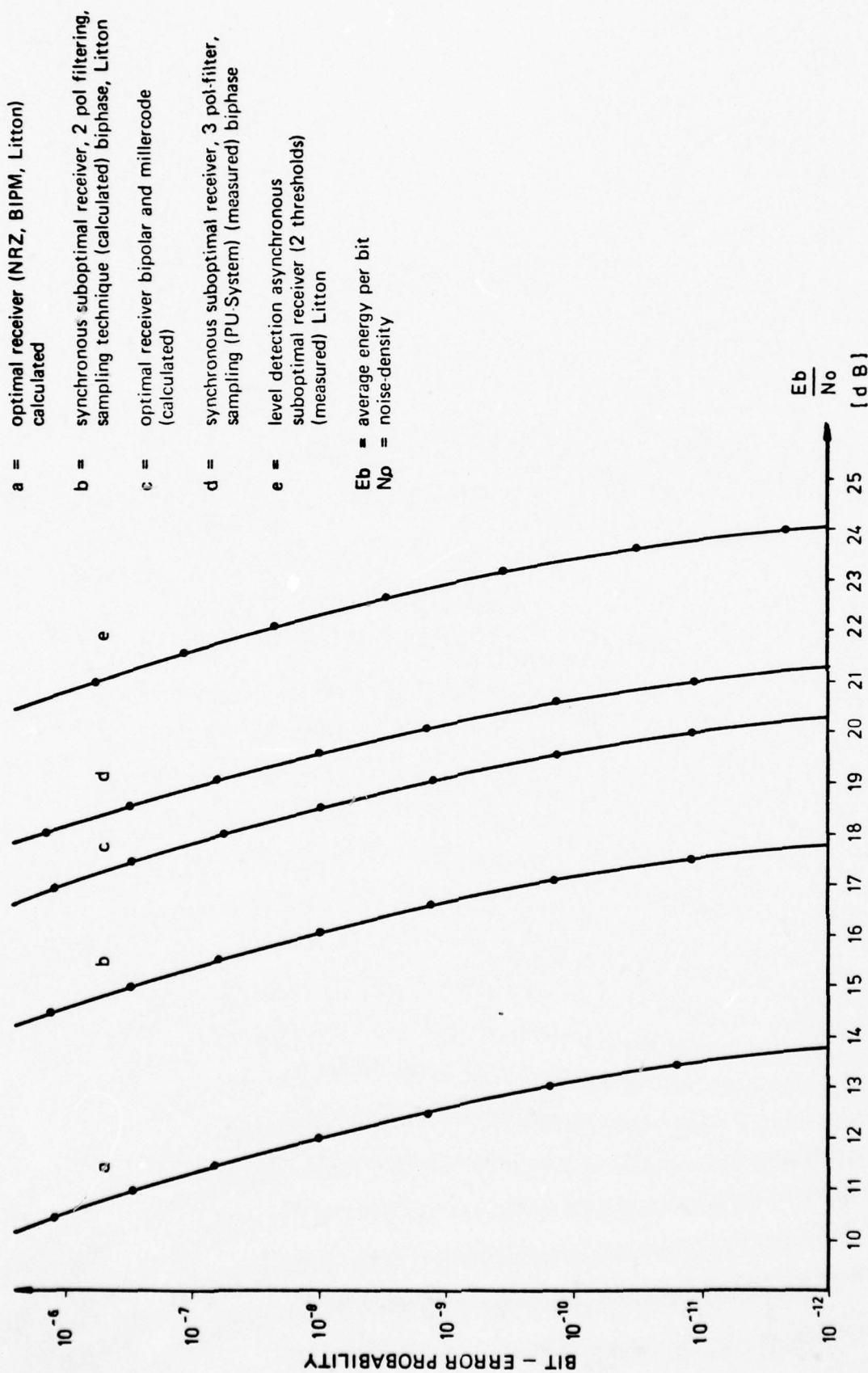


Fig.18 Comparison of receiver methods

ANNEX B

MILITARY STANDARD 1553A

CONTENTS

	Page
1. SCOPE AND PURPOSE	B-5
1.1 Scope	B-5
1.2 Purpose	B-5
2. APPLICABLE DOCUMENTS	B-5
3. DEFINITIONS	B-5
3.1 Remote Terminal	B-5
3.2 Bit	B-6
3.3 Bit Rate	B-6
3.4 Pulse Code Modulation (PCM)	B-6
3.5 Time Division Multiplexing (TDM)	B-6
3.6 Command/Response Mode	B-6
3.7 Half Duplex	B-6
3.8 Asynchronous Operation	B-6
3.9 Dynamic Bus Allocation	B-6
3.10 Word	B-6
3.11 Message	B-6
3.12 Data Bus	B-6
3.13 Controller	B-6
4. REQUIREMENTS	B-6
4.1 Data Bus Operation	B-6
4.1.1 Information Transfer Modes	B-6
4.2 Characteristics	B-6
4.2.1 Data Form	B-6
4.2.2 Bit Priority	B-6
4.2.3 Transmission Method	B-7
4.2.3.1 Modulation	B-7
4.2.3.2 Data Code	B-7
4.2.3.3 Transmission Rate	B-7
4.2.3.4 Word Size	B-7
4.2.3.5 Word Formats	B-7
4.2.3.5.1 Command Word	B-7
4.2.3.5.1.1 Sync	B-7
4.2.3.5.1.2 Address	B-7
4.2.3.5.1.3 Transmit/Receive	B-7
4.2.3.5.1.4 Subaddress/Mode	B-7
4.2.3.5.1.5 Word Count	B-8
4.2.3.5.1.6 Parity	B-8
4.2.3.5.1.7 Operational Mode Control	B-8
4.2.3.5.2 Data Word	B-8
4.2.3.5.2.1 Sync	B-8
4.2.3.5.2.2 Data	B-8
4.2.3.5.2.3 Parity	B-8
4.2.3.5.3 Status Word	B-8
4.2.3.5.3.1 Sync	B-9
4.2.3.5.3.2 RT Address	B-9
4.2.3.5.3.3 Message Error	B-9
4.2.3.5.3.4 Status Codes	B-9
4.2.3.5.3.5 Terminal Flag	B-9
4.2.3.5.3.6 Parity	B-9
4.2.3.6 Message Formats	B-9
4.2.3.6.1 Controller to RT Transfers	B-9
4.2.3.6.2 RT to Controller Transfers	B-9
4.2.3.6.3 RT to RT Transfers	B-10
4.2.4 Transmission Line	B-10
4.2.4.1 Cable	B-10
4.2.4.2 Characteristic Impedance	B-10
4.2.4.3 Cable Attenuation	B-10
4.2.4.4 Cable Length	B-10
4.2.4.5 Cable Termination	B-10

		Page
4.2.4.6	Cable Coupling	B-10
4.2.4.7	Wiring and Cabling for EMC	B-10
4.2.5	RT/Bus Interface Circuits	B-11
4.2.5.1	Circuit Configuration	B-11
4.2.5.2	Fault Isolation	B-11
4.2.5.3	RT Output Characteristics	B-11
4.2.5.3.1	Output Levels	B-11
4.2.5.3.2	Output Waveform	B-11
4.2.5.3.3	Output Noise	B-11
4.2.5.4	RT Input Characteristics	B-11
4.2.5.4.1	Input Waveform Compatibility	B-11
4.2.5.4.2	Common Mode Rejection	B-11
4.2.5.4.3	Input Impedance	B-11
4.2.5.4.4	Data Validation	B-12
4.3	Terminal Operation	B-12
4.3.1	Response Time	B-12
4.3.2	Terminal Fail-Safe Operation	B-12
4.3.3	Noise Environment Operation	B-12
4.3.3.1	Test Environment	B-12
4.3.3.1.1	Electric Field	B-12
4.3.3.1.2	Magnetic Field	B-12
4.3.3.2	Bit Error Rate	B-12
4.3.3.3	Incomplete Message Rate	B-12
4.3.3.4	Test Conditions	B-12
4.4	Terminal to Subsystem Interface	B-12
4.4.1	Serial Digital Interface	B-13
4.4.1.1	Serial Digital Input	B-13
4.4.1.2	Serial Digital Output	B-14
4.4.1.3	Signal Characteristics	B-16
4.4.2	Discrete Signals	B-16
4.5	Bus Controller	B-16

APPENDIX

10	GENERAL	B-17
10.1	Redundancy	B-17
10.2	Bus Controller	B-17
10.3	Multiplex Selection Criteria	B-17
10.4	High Reliability Requirements	B-18
10.5	Stubbing	B-18
10.6	Status Code Usage	B-18
10.6.1	Vectored Service Request	B-18
10.6.2	Error Code Supplement	B-18
10.6.3	Assigned Codes	B-18

FIGURES

	Page
1 Typical multiplex data bus architecture	B-5
2 Data encoding	B-7
3 Word formats	B-8
4 Command and status sync	B-8
5 Data sync	B-9
6 Message formats	B-9
7 Data bus interface	B-10
8 Output waveform	B-11
9 Serial digital interface	B-13
10 Serial digital input interface	B-14
11 Serial digital input interface timing diagram for external initiation of transfer	B-14
12 Serial digital input interface timing diagram for terminal initiation of transfer	B-15
13 Serial digital output interface	B-15
14 Serial digital output interface timing diagram	B-15
10.1 Illustrations of possible redundancy	B-17
10.2 Illustrations of possible redundancy	B-17

TABLE

1 Signal Definitions for Serial Digital Interfaces	B-13
--	------

1. SCOPE AND PURPOSE

1.1 Scope. This standard defines requirements for digital, command/response, time division multiplexing (Data Bus) techniques on aircraft. It encompasses the data bus line and its interface electronics as illustrated on Figure 1, and also defines the concept of operation and information flow on the multiplex data bus and the electrical and functional formats to be employed.

1.2 Purpose. The purpose of this document is to establish uniform requirements for multiplex data system techniques which will be utilized in systems integration of aircraft subsystems and to promote standard digital interfaces for associated subsystems. The system designer retains the flexibility to assemble a custom multiplex system from the functionally standard parts and to program the standard electronic functions in order to provide a control mechanism, traffic patterns, redundancy, and a viable degradation concept.

2. APPLICABLE DOCUMENTS

2.1 The following documents, of the issue in effect on date of invitation for bid or request for proposal, form a part of the standard to the extent specified herein.

Specification

Military

MIL-E-6051 Electromagnetic Compatibility Requirements, Systems

Standards

Military

MIL-STD-461 Electromagnetic Interference Characteristics, Requirements for Equipment

MIL-STD-462 Electromagnetic Interference Characteristics, Measurement of

(Copies of specifications, standards, drawings, and publications required by suppliers in connection with specific procurement functions should be obtained from the procuring activity or as directed by the contracting officer.)

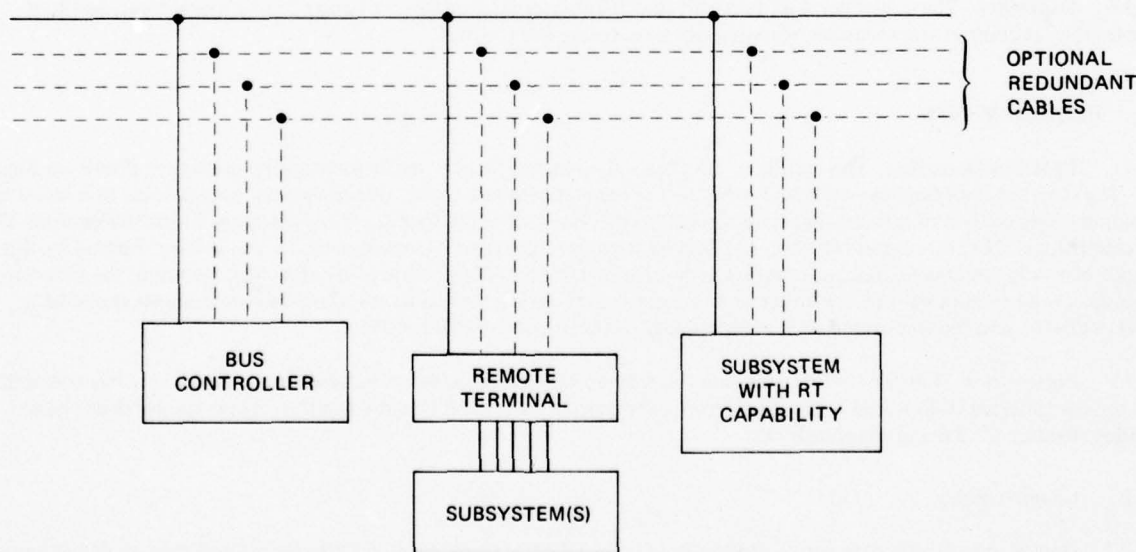


Fig.1 Typical multiplex data bus architecture

3. DEFINITIONS

3.1 Remote Terminal. The remote terminal is the electronics necessary to interface the bus with the subsystem and the subsystem with the bus. This electronics may exist as a separate LRU (line replaceable unit), or be contained within the users' subsystem. A redundant bus controller, when not functioning as a controller, may operate as a remote terminal.

3.2 Bit. Contraction of binary digit: may be either zero or one. In information theory a binary digit is equal to one binary decision or the designation of one of two possible values or states of anything used to store or convey information.

3.3 Bit Rate. The number of bits transmitted per second.

3.4 Pulse Code Modulation (PCM). The form of modulation in which the modulation signal is sampled, quantized, and coded so that each element of information consists of different types or numbers of pulses and spaces.

3.5 Time Division Multiplexing (TDM). The transmission of information from several signal channels through one communication system with different channel samples staggered in time to form a composite pulse train.

3.6 Command/Response Mode. The operation of a bus system in which the remote terminal will respond only when commanded by the bus controller.

3.7 Half Duplex. Operation of a data transfer system in either direction over a single line, but not in both directions on that line simultaneously.

3.8 Asynchronous Operation. For the purpose of this standard, asynchronous bus operation implies an independent clock source at each remote terminal which is utilized for the transmission of messages. The received message shall be decoded using clock information derived from the received signal.

3.9 Dynamic Bus Allocation. The operation of a bus system in which designated remote terminals are offered control of the data bus.

3.10 Word. In this document a word is a sequence of 16 bits plus sync and parity. There are three types of words: command, status and data.

3.11 Message. A message is a transmission of words on the data bus cable. A message transfer is complete when the command word, data word(s) and the status word have been transmitted. There are three types of messages: controller to terminal, terminal to controller and terminal to terminal.

3.12 Data Bus. Whenever a data bus or bus is referred to in this document it shall imply a single twisted shielded pair cable.

3.13 Controller. The controller shall be a unit that is either programmable, or controlled by a processor, and that serves the function of commanding, scanning and monitoring bus traffic.

4. REQUIREMENTS

4.1 Data Bus Operation. The multiplex data bus in its most elemental configuration shall operate as shown on Figure 1. The data bus shall function asynchronously in a command/response mode, and transmission shall occur in a half-duplex manner. Sole control of information transmission on the bus shall reside with the bus controller, which shall initiate all transmissions. The information flow on the data bus shall be comprised of messages which are, in turn, formed by three types of words (command, data, and status) as defined in 4.2.3.5. All elements of the data bus, including the transmission line, remote terminal and controller, shall conform to the electromagnetic interference requirements specified in MIL-STD-461 and the electromagnetic compatibility requirements of MIL-E-6051.

4.1.1 Information Transfer Modes. The data bus may employ three modes of information transfer: (1) bus controller to remote terminal (RT) transfer, (2) RT to controller transfer and (3) RT to RT transfer. These modes shall operate as described in 4.2.3.6 and subparagraphs.

4.2 Characteristics

4.2.1 Data Form. Digital data may be transmitted in any desired form, provided that the chosen form shall be compatible with the message and word formats defined in this standard. Any unused bit positions in a word shall be transmitted as logic zeros.

4.2.2 Bit Priority. The most significant bit shall be transmitted first with the less significant bits following in descending order of value. The number of bits required to define a quantity shall be consistent with the resolution or accuracy required. In the event double precision quantities (information accuracy or resolution requiring more than 16 bits) are transmitted, the more significant half shall be transmitted first, followed by the less significant half.

4.2.3 Transmission Method

4.2.3.1 *Modulation.* The signal shall be transferred over the data bus in serial digital pulse code modulation form.

4.2.3.2 *Data code.* The data code shall be Manchester bi-phase level. A logic one shall be transmitted as a bipolar coded signal 1/0 (i.e., a positive pulse followed by a negative pulse). A logic zero shall be bipolar coded signal 0/1 (i.e., a negative pulse followed by a positive pulse). A transition through zero occurs at the midpoint of each bit time (see Figure 2).

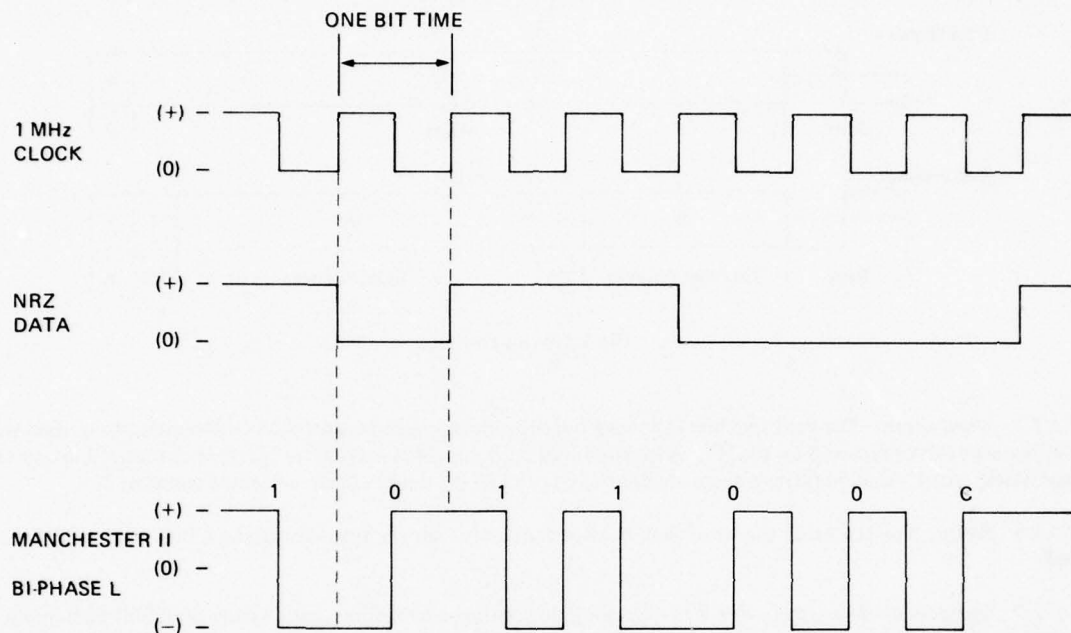


Fig.2 Data encoding

4.2.3.3 *Transmission rate.* The transmission rate on the bus shall be 1.0 megabit per second with a long term stability of ± 0.01 percent (i.e., ± 100 Hz). The short term stability (i.e., stability over a 1.0 second interval) shall be at least 0.001 percent (i.e., ± 10 Hz).

4.2.3.4 *Word size.* The word size shall be 16 bits plus the sync wave form and the parity for a total of 20 bit times as shown in Figure 3.

4.2.3.5 *Word formats.* The word formats shall be as shown on Figure 3 for the command, data, and status words.

4.2.3.5.1 *Command word.* A command word shall be comprised of a sync waveform, address, transmit/receive bit, subaddress/mode, data word count, and a parity bit (see Figure 3), except as modified by 4.2.3.5.1.7.

4.2.3.5.1.1 *Sync.* The command sync waveform shall be an invalid Manchester waveform as shown on Figure 4. The width shall be three bit times, with the waveform being positive for the first one and one-half bit times, and then negative for the following one and one-half bit times. If the next bit following the sync is a logic zero, then the last half of the sync waveform will have an apparent width of two clock periods due to the Manchester encoding.

4.2.3.5.1.2 *Address.* The next five bits following the sync shall be the RT address. This permits a maximum of 32 RTs to be attached to any one data bus. All 1's shall indicate a decimal address of 31, and all 0's shall indicate a decimal address of 32. The most significant bit of the address shall be transmitted first.

4.2.3.5.1.3 *Transmit/receive.* The next bit following the address shall be the transmit/receive bit, which shall indicate the action required of the RT. A logic zero shall indicate the RT is to receive, and a logic one shall indicate the RT is to transmit.

4.2.3.5.1.4 *Subaddress/mode.* The next five bits following the transmit/receive bit shall be utilized for either a RT subaddress or mode control, as is dictated by the individual terminal requirements. The subaddress/ mode value of 00000 is reserved for special purposes, as specified in 4.2.3.5.1.7, and shall not be utilized for any other function.

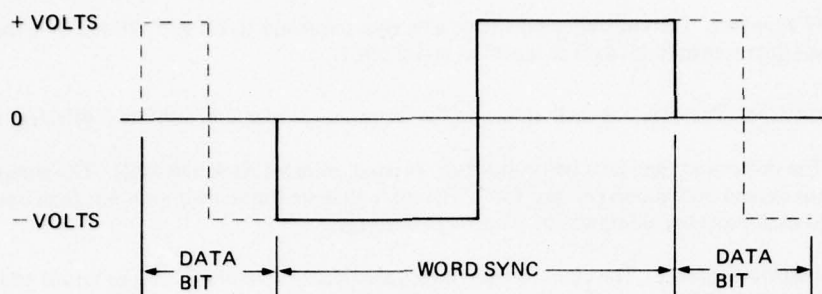


Fig.5 Data sync

4.2.3.5.3.1 *Sync*. The sync waveform shall be as specified in 4.2.3.5.1.1.

4.2.3.5.3.2 *RT address*. The next five bits following the sync shall contain the address of the terminal which is transmitting the status word as defined in 4.2.3.5.1.2.

4.2.3.5.3.3 *Message error*. The first bit after the address shall be utilized to indicate that the preceding message failed to pass the RT's validity test. This error condition shall include parity errors. A logic one shall indicate the presence of a message error, and a logic zero its absence. A message error shall be indicated when the preceding message to a RT has failed either the word or message validity criteria for the RT. The criteria shall include those specified in 4.2.5.4.4.

4.2.3.5.3.4 *Status codes*. The next nine bits following the message error bit may be utilized in any fashion desired by the RT designer, except that all zeros shall indicate a normally functioning terminal.

4.2.3.5.3.5 *Terminal flag*. The next to least significant bit in the status word is reserved for a terminal flag bit. This bit shall be set to one to indicate the need for the bus controller to examine the built in test data available from the terminal.

4.2.3.5.3.6 *Parity*. The last bit shall be utilized for parity as specified in 4.2.3.5.1.6.

4.2.3.6 *Message formats*. The messages transmitted on the data bus shall be in accordance with the formats in Figure 6. The maximum and minimum response times shall be as stated in 4.3.1.

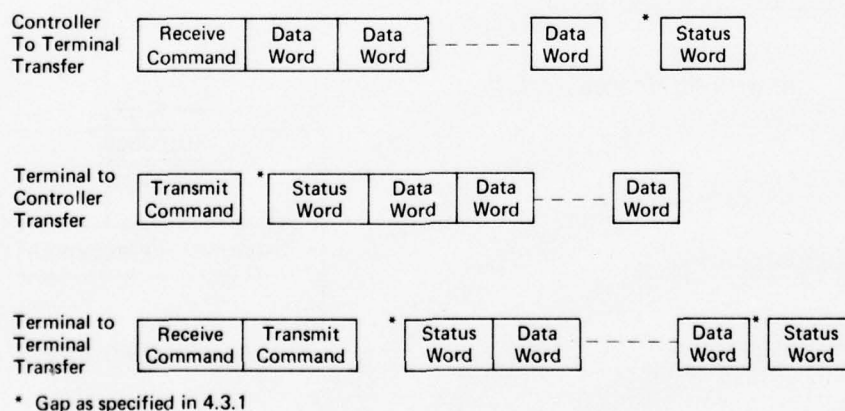


Fig.6 Message formats

4.2.3.6.1 *Controller to RT transfers*. The controller shall issue a receive command followed by the specified number of data words. The RT shall, after message validation, transmit a status word back to the controller. The command and data words shall be transmitted in a continuous fashion with no interword gaps.

4.2.3.6.2 *RT to controller transfers*. The controller shall issue a transmit command to the RT. The RT shall, after command verification, transmit a status word back to the controller, followed by the specified number of data words. The status and data words shall be transmitted in a continuous fashion with no interword gaps.

4.2.3.6.3 *RT to RT transfers.* The controller shall issue a receive command to RT A, followed by a transmit command to RT B. RT B shall then transmit the data as specified in 4.2.3.6.1.

4.2.4 *Transmission Line.* The data bus shall utilize, as the transmission medium, a twisted, shielded, wire pair.

4.2.4.1 *Cable.* The cable used shall be a two conductor, twisted, shielded, jacketed cable. The wire-to-wire distributed capacitance shall not exceed 30.0 picofarads per foot. The cable shall be formed with not less than one twist per inch; and the cable shield shall provide a minimum of 80 percent coverage.

4.2.4.2 *Characteristic impedance.* The characteristic impedance shall be 70 ohms, plus or minus 10 percent, at a sinusoidal frequency of 1.0 MHz.

4.2.4.3 *Cable attenuation.* At the frequency of 4.2.4.2, the cable power loss shall be 1 db/100 ft or less.

4.2.4.4 *Cable length.* The cable length of any main bus may be up to 300 feet.

4.2.4.5 *Cable termination.* The cable shall be coupled to the RT as shown on Figure 7. A long stub is defined as any stub greater than one foot in length. The use of long stubs is discouraged and the length of any stub shall not exceed 20 feet. The two ends of the cable shall be terminated with a resistance equal to the cable characteristic impedance.

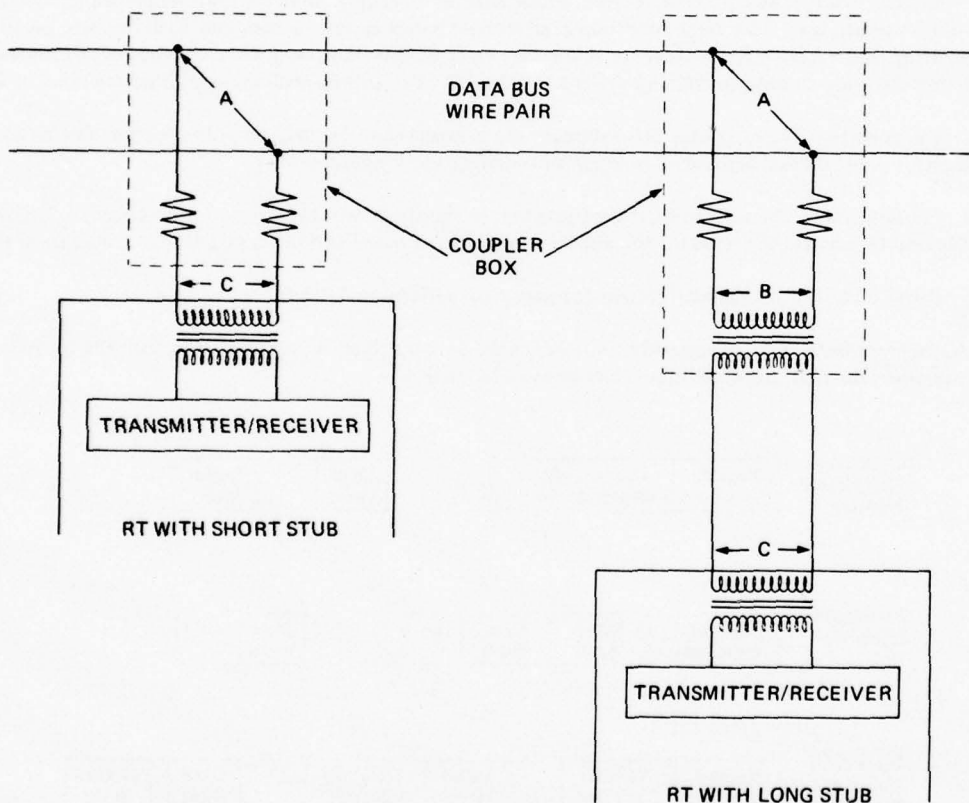


Fig.7 Data bus interface

4.2.4.6 *Cable coupling.* All connections to the data bus shall utilize a small shielded coupler box. This box shall be of sufficient size to permit the installation of the transformer and isolation resistors specified in 4.2.5. The connector plug shall be compatible with Amphenol Type 31-235 or Trompeter Type TEI-14949-EI37 receptacles. The connector receptacle shall be compatible with Amphenol Type 31-224 or Trompeter Type TEI-14949-PL36 plugs. The polarity convention shall be that the female connection in the plug is positive, and the male connection in the receptacle is positive. This connector, with the indicated polarities, shall be used for all bus interfaces.

4.2.4.7 *Wiring and cabling for EMC.* For purposes of electromagnetic compatibility (EMC), the wiring and cabling provisions of MIL-E-6051 shall apply.

4.2.5 RT/Bus Interface Circuits

4.2.5.1 Circuit configuration. The input/output circuits shall consist of a transmitter-receiver, DC isolation/coupling transformer, and isolation resistors as configured on Figure 7.

4.2.5.2 Fault isolation. An isolation resistor shall be placed in series with each connection to the data bus cable. This resistor shall have a value of $0.75 Z_0$ ohms plus or minus 5 percent where Z_0 is the cable characteristic impedance. The impedance placed across the data bus cable shall be no less than $1.5 Z_0$ ohms for any failure of the coupling transformer, cable stub, or RT transmitter/receiver.

4.2.5.3 RT output characteristics

4.2.5.3.1 Output levels. The RT signal output circuitry shall be capable of driving the cable specified in 4.2.4.1 and not less than 33 other RTs, as specified herein, each attached to the cable by means of a cable stub of maximum length specified in 4.2.4.5. The output circuitry shall maintain the specified operation with the exception of a 25 percent maximum reduction of the data bus signal amplitude in the event that one of the RTs has a fault that causes it to reflect the fault impedance specified in 4.2.5.2 on the bus. The RT peak signal output voltage shall be between plus or minus 3.0 and 10.0 volts, line-to-line, when measured at the data bus cable connection (point A on Figure 7).

4.2.5.3.2 Output waveform. The waveform when observed at point C in Figure 7 shall have zero crossings which deviate not more than plus or minus 25 nanoseconds from those shown in Figure 8. The rise and fall time of this waveform shall be equal to or greater than 100 nanoseconds when measured from the levels of 10 to 90 percent of full waveform peak-to-peak voltage, as shown in Figure 8.

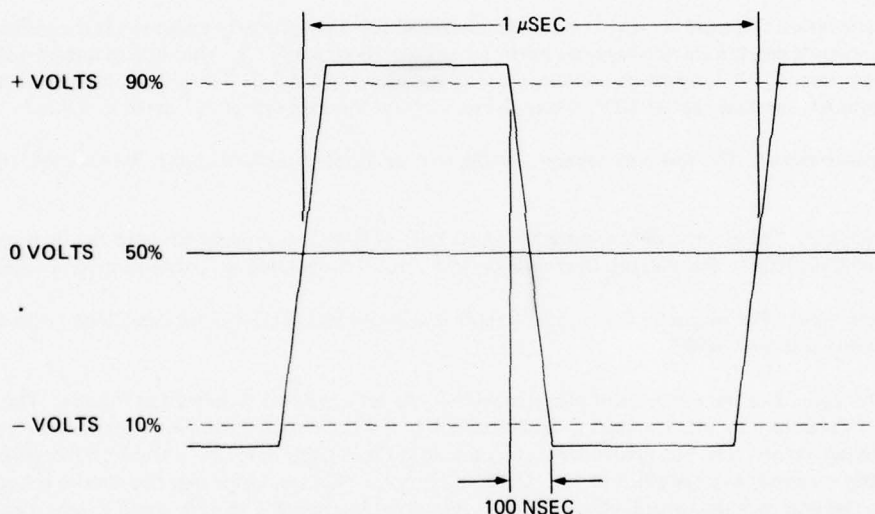


Fig.8 Output waveform

4.2.5.3.3 Output noise. Any noise transmitted to the data bus when the RT is receiving or has power removed, shall not exceed a value of 10.0 millivolts peak-to-peak, line-to-line, as measured at the point specified in 4.2.5.3.1.

4.2.5.4 RT input characteristics

4.2.5.4.1 Input waveform compatibility. The RT shall be capable of receiving and operating with the incoming signals specified herein, and shall accept waveforms varying from a square wave to a sine wave. The RT shall respond to an input signal whose positive or negative peak amplitude, line-to-line, is within the range of 10.0 to 0.5 volts. The voltages are measured at point C in Figure 7.

4.2.5.4.2 Common mode rejection. Any signals from dc to 2.0 MHz, with amplitudes equal to or less than plus or minus 10.0 volts peak, line-to-ground, applied to point A as shown in Figure 7 shall not degrade the performance of the RT.

4.2.5.4.3 Input impedance. The magnitude of the RT input impedance, when the RT is not transmitting, or has power removed, shall be a minimum of 2000 ohms within the frequency range of 100 KHz to 1.0 MHz. This impedance is that measured line-to-line at point C on Figure 7.

4.2.5.4.4 Data validation. Logic shall be provided in each RT to recognize improperly coded signals, data dropouts, or excessively noisy signals. Each word shall conform to the following minimum validating criteria:

- (a) The word begins with a valid sync field.
- (b) The bits are in a valid Manchester II code.
- (c) The information field has 16 bits plus parity.
- (d) The word parity is odd.

Where a word fails to conform to the preceding criteria, the word shall be considered invalid and shall not be used by the receiving RT. Where an invalid word sync occurs, the receiving RT shall reset and wait for a new valid message sync. An invalid word count shall be construed as a message transmission error.

4.3 Terminal Operation. The remote terminal shall operate in response to commands received from the bus controller. The RT shall be capable of receiving a command word at any time except when it is transmitting. A second command word sent to a terminal after it is already operating on one shall invalidate the first command and cause the RT to begin operation on the second command.

4.3.1 Response Time. The RT shall respond to a valid transmit data command during the time period 2.0 to 5.0 microseconds after receipt of the last bit of the command word. The RT shall respond to a valid receive data command during the time period 2.0 to 5.0 microseconds after receipt of the last bit of the last data word.

4.3.2 Terminal Fail-Safe Operation. The RT shall contain the self-test circuitry necessary to detect an erroneous transmission of data on to the data bus. This circuitry shall include a transmission time-out which will preclude a signal transmission period of greater than 660 microseconds (one status, and thirty-two data words). When the self-test circuitry detects any such erroneous transmission, it shall automatically shut down the transmitter portion of the RT.

4.3.3 Noise Environment Operation. The remote terminal shall function properly under the test conditions specified in 4.3.3.4, and encountering the electromagnetic environment specified in 4.3.3.1. The remote terminal shall exhibit a maximum bit error rate of 10^{-12} , where the bit error rate is as defined in 4.3.3.2. The remote terminal shall also exhibit a maximum incomplete message rate of 10^{-6} , where the incomplete message rate is as defined in 4.3.3.3.

4.3.3.1 Test environment. The test environment for the remote terminal and data bus cable radiated susceptibility shall be as follows:

4.3.3.1.1 Electric field. The electric field test shall employ MIL-STD-462 method RS03, with the limit specified in MIL-STD-461 test limit RS03. The electric field shall be 100 percent modulated by a waveform as specified in 4.2.3.

4.3.3.1.2 Magnetic field. The magnetic field (spike test) shall employ MIL-STD-462 method RS02, with the limit specified in MIL-STD-461 test limit RS02.

4.3.2.2 Bit error rate. For the purposes of paragraph 4.3.3, the bit error rate is defined as follows: The bus controller transmits 32 data words to a remote terminal as specified in 4.1, and the remote terminal responds with a status word indicating no message errors. The bus controller then commands the remote terminal to transmit the same 32 data words which it previously received, as is specified in 4.1. Upon receipt of a valid response from the remote terminal, the controller then compares each data word which it sent to the remote terminal with each one it received back from the remote terminal. The sixteen bits in each word pair are compared and if any bit does not match, this is to be considered a bit error. The total number of data bits transmitted during a specific time period are counted. The bit error rate is then defined as the number of bit errors, divided by the total number of bits transmitted.

4.3.3.3 Incomplete message rate. For the purposes of paragraph 4.3.3, the incomplete message rate is defined as follows: A message is the set of command, data, and status words as defined in 4.2.3.6. An incomplete message is defined as one during which the remote terminal does not properly respond to a command by the bus controller, or one in which the message error bit is set in the remote terminal status word. The total number of incomplete messages are counted during a specific time period, as are the total number of messages. The incomplete message rate is given by the number of incomplete messages divided by the total number of messages. The message error bit in the first status word following a non-reponse by a remote terminal shall not be included in the incomplete message count. The message formats shall be as defined in 4.3.3.2.

4.3.3.4 Test conditions. For purposes of the noise tests, the following conditions shall be observed. All data words shall be changed to random bit patterns prior to each transmission/reception set as defined in 4.3.3.2. The test shall be conducted with the bus controller and the remote terminal both connected by 20 foot stubs to the main data bus cable, with a minimum distance of 100 feet between the stubs. The remote terminal transmitter shall provide an output as specified in 4.2.5.3. The bus controller transmitter shall have its output adjusted so as to provide the minimum signal amplitude specified in 4.2.5.4.1 at the remote terminal.

4.4 Terminal to Subsystem Interface. For those applications where a terminal is not contained within a subsystem, and the terminal exists as a distinct LRU, the terminal shall provide the necessary electronics to interface to the

subsystem. The terminal shall have provisions for the standard serial digital and discrete interfaces as defined in the following paragraphs. All other signals shall require special purpose interface provisions within the terminal, this electronics being designed for the peculiar interface requirement.

4.4.1 Serial Digital Interface. The standard serial digital interface shall be configured as shown in Figure 9. All lines are unidirectional, with the data line's direction to be determined by its usage, i.e., to transmit or to receive data. The interface shall operate as defined in 4.4.1.1 for an input interface and as defined in 4.4.1.2 for an output interface. The functions of each of the signals is as defined in Table I.

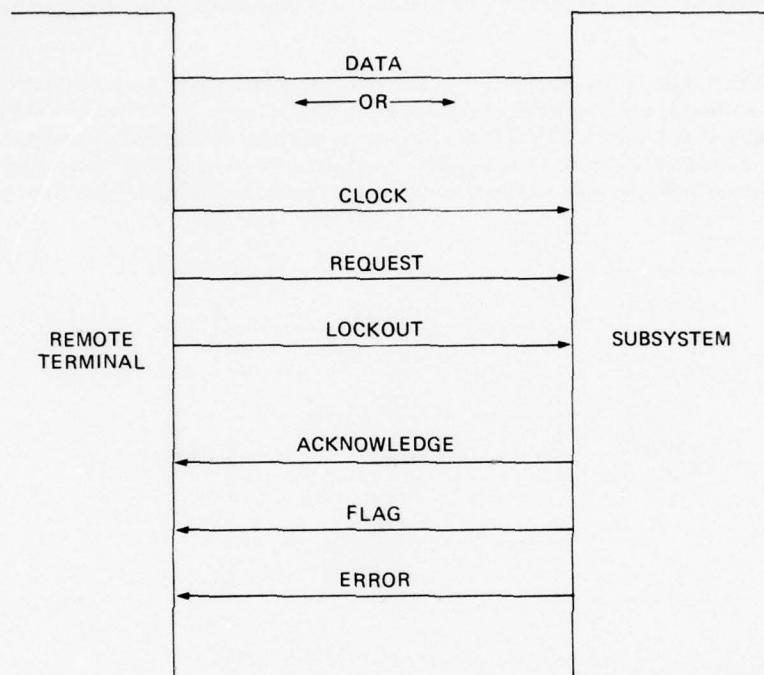


Fig.9 Serial digital interface

4.4.1.1 Serial digital input. A serial digital input interface is a set of six signals between an external device and the remote terminal. This interface is shown in Figure 10. The function of each of the six signals is defined in Table I. The performance of a data input sequence can be initiated by either of two actions. The external device can pulse the FLAG line if the LOCKOUT line is low, and thereby initiate a data input sequence. The timing diagram for this action is shown in Figure 11. The bus controller can directly command the remote terminal to begin a data input sequence. The timing diagram for this action is shown in Figure 12. In either case, the initiation of the data input sequence causes the LOCKOUT line to be set, and the completion of the data input sequence shall cause the remote terminal to notify the bus controller of the data input, and of any parity errors. The remote terminal shall be required to clear the LOCKOUT before any new externally initiated data input sequences can occur.

TABLE I

Signal Definitions for Serial Digital Interfaces

- 1. REQUEST.** This is a signal from the remote terminal to the external device which, when set to logic 1 by the remote terminal, notifies the external device that a data transfer is about to take place, and when set to logic 0 by the remote terminal, notifies the external device that a data transfer is complete.
- 2. ACKNOWLEDGE.** This is a signal from the external device to the remote terminal which, when set to logic 1 by the external device, notifies the remote terminal that the external device has recognized the REQUEST, and is ready for the data transfer, and when set to logic 0 by the external device, notifies the remote terminal that the external device has recognized the lowering of the REQUEST and the end of the data transfer.
- 3. CLOCK.** This is a signal from the remote terminal to the external device, which when active is a 1 MHz square wave, with a number of cycles equal to the number of bits to be shifted. The CLOCK is not started until the remote terminal has seen the ACKNOWLEDGE line raise. At the end of the last CLOCK cycle the remote terminal shall lower the REQUEST Line.

TABLE I (Continued)

4. **DATA.** This is a signal to or from the remote terminal upon which data is transmitted. On the positive edge of the **CLOCK** signal the next **DATA** bit shall be placed on the **DATA** line.
5. **LOCKOUT.** This is a signal from the remote terminal to the external device which, when set to logic 1 by the remote terminal, notifies the external device that it shall refuse all external requests for data transfer, and when set to logic 0 by the remote terminal, notifies the external device that it shall allow external requests for data transfer.
6. **ERROR.** This is a signal from the external device to the remote terminal which the external device clears at the time **ACKNOWLEDGE** is raised, and which the external device sets at any time that a parity error is detected while receiving data.
7. **FLAG.** This is a signal from the external device to the remote terminal, which is a pulse of duration between 1 and 10 microseconds, that notifies the remote terminal that there has been an external request for a **DATA** input sequence. The occurrence of this pulse shall initiate a **DATA** input sequence, and after the sequence is completed the bus controller shall be informed that the sequence occurred and whether or not there were any parity errors. This signal cannot occur while the **LOCKOUT** line is high, and the occurrence of this signal causes the **LOCKOUT** line to be set high.

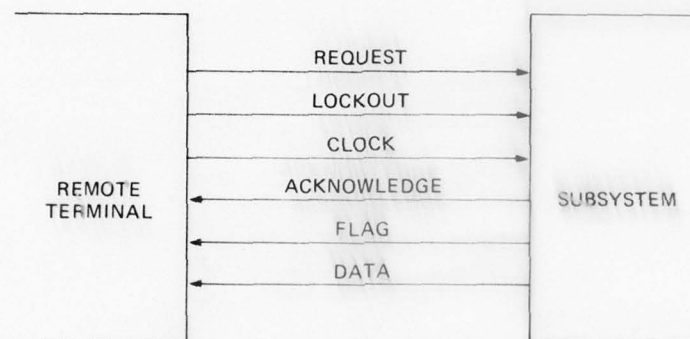


Fig. 10 Serial digital input interface

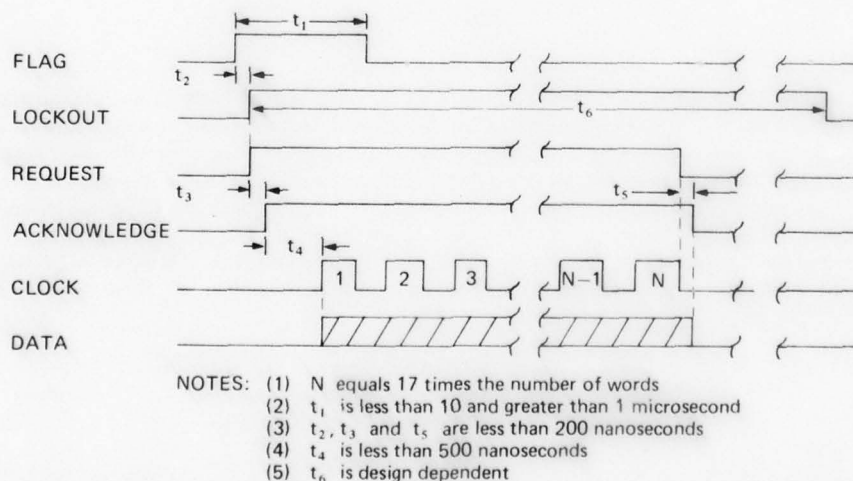
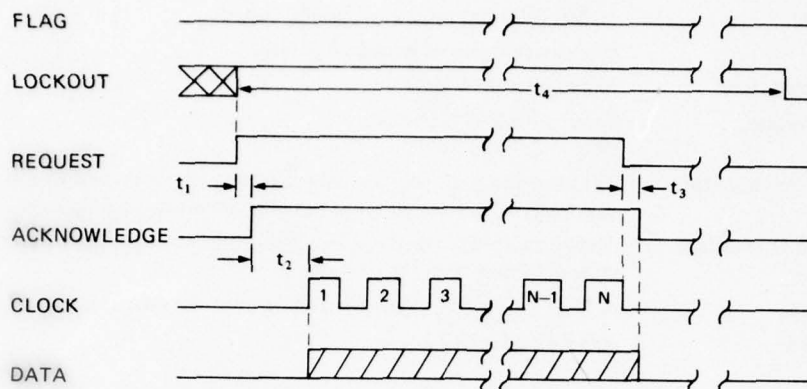


Fig. 11 Serial digital input interface timing diagram for external initiation of transfer

4.4.1.2 Serial digital output. A serial digital output interface is a set of five signals between an external device and the remote terminal. This interface is shown in Figure 13. The functions of each of these five signals is defined in Table I. The timing diagram for the data output sequence using these five signals is shown in Figure 14. The performance of a data output sequence can be initiated by either of two bus controller actions. The bus controller can send a new data block to the remote terminal, and the receipt of this data block shall initiate a data output sequence. The bus controller can also directly command the remote terminal to begin a data output sequence using the data block that the remote

terminal has available. In either case, once a data output sequence is initiated, the serial interface shall always transfer the complete set of data, and when the data transfer is complete the remote terminal shall examine the ERROR line.



- NOTES: (1) N equals 17 times the number of words
 (2) t_1 and t_3 are less than 200 nanoseconds
 (3) t_2 is less than 500 nanoseconds
 (4) t_4 is design dependent

Fig.12 Serial digital input interface timing diagram for terminal initiation of transfer

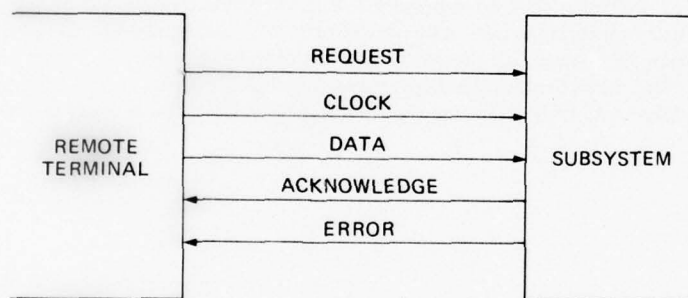
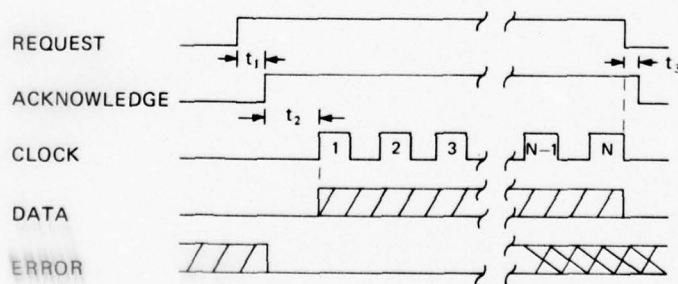


Fig.13 Serial digital output interface



- NOTES: (1) N equals 17 times the number of words
 (2) ERROR line is based on parity check by subsystem
 (3) t_1 and t_3 are less than 200 nanoseconds
 (4) t_2 is less than 500 nanoseconds

Fig.14 Serial digital output interface timing diagram

4.4.1.3 Signal characteristics. The characteristics of serial digital signals shall be in accordance with the following:

- | | |
|--------------------------------------|--|
| (a) Data code | Non-return-to zero (NRZ) |
| (b) Type | Differential and balanced |
| (c) Data word | 16 bits followed by one bit of odd parity |
| (d) Data rate | One megabit plus or minus 10 percent |
| (e) Rise and fall time | As specified in 4.2.5.3.2 |
| (f) Output voltage | Zero: -0.5 to 0.5 volts
One: 2.4 to 5.5 volts |
| (g) Common mode output voltage | The common mode output voltage (measured from each line to the signal common) of the output circuit shall be no greater than plus or minus 0.5 volt peak |
| (h) Short and Overvoltage protection | The output circuit shall not be damaged when subjected to shorts to ground or a voltage of plus or minus 20 volts |
| (i) Message size | A fixed number of words for each request with a maximum of 32 words |
| (j) Bit priority | As specified in 4.2.2. |

4.4.2 Discrete Signals. The discrete interface shall be double-ended, and shall employ the following logic levels:

- Zero: -0.5 to 0.5 volts
One: 2.4 to 5.5 volts

The input circuits shall present a minimum impedance of 10K ohms. Overvoltage faults to an input of up to plus or minus 20 volts shall not damage the input. The output circuits shall be capable of providing a minimum output current of 100 milliamperes. Short circuits on either inputs or outputs shall not damage the circuits.

4.5 Bus Controller. The controller shall be responsible for sending data bus commands, participating in data transfer, receiving status response and monitoring system status as defined in this standard. The controller may be embodied as either a stand alone hardware unit whose sole function is to control the data bus(s), or contained within the I/O section of an airborne computer. The controller shall be programmable and shall operate under software (or firmware) control. Individual application requirements shall determine the choice as to which form of controller is used.

APPENDIX

10. GENERAL

The following paragraphs in this appendix are presented in order to discuss certain aspects of the standard in a general sense. They are intended to provide a user of the standard more insight into the aspects discussed.

10.1 Redundancy. It is intended that this standard be used to support rather than to supplant the system design process. For this reason, the standard is deliberately vague concerning the use of redundancy in implementing a multiplex data bus system. The system designer should utilize this standard as the needs of a particular application dictate. The use of redundancy, the degree to which it is implemented, and the form which it takes must be determined on an individual application basis. Figures 10.1 and 10.2 illustrate some possible approaches to dual redundancy.

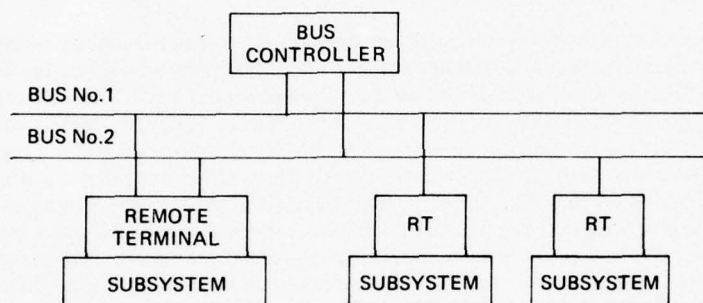
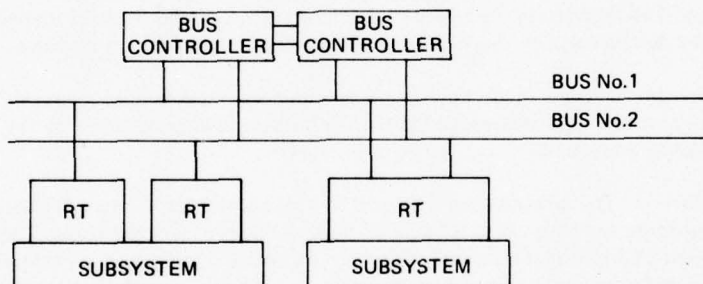


Fig.10.1



NOTE: RT - Remote Terminal

Fig.10.2

Fig.10 Illustrations of possible redundancy

10.2 Bus Controller. The bus controller is a key part of the data bus system. The functions of the controller, in addition to the issuance of commands, must include the constant monitoring of the data bus and the traffic on the bus. It is envisioned that most of the routine minute details of bus monitoring (e.g., parity checking, terminal non-response time-out, etc.) will be embodied in hardware, while the algorithms for bus control and decision making will reside in software. It is also envisioned that, in general, the bus controller will be a general purpose airborne computer with a special input/output (I/O) unit to interface with the data bus. In the case of a large aircraft, such as a bomber, the multiplex bus control problem may be of sufficient complexity to warrant the employment of a dedicated bus controller. While in a smaller, fighter-type aircraft, the control function will probably be incorporated into a computer which is also utilized for navigation and weapon delivery functions. It is important to remember that the controller will be the focal point for modification and growth within the multiplex system, and thus the software must be written in such a manner as to permit modification with relative ease.

10.3 Multiplex Selection Criteria. The selection of candidate signals for multiplexing is a function of the particular application involved, and criteria will in general vary from system to system. Obviously those signals which have bandwidths of 400 Hz or less are prime candidates for inclusion on the bus. It is also obvious that video, audio, and high speed parallel digital signals should be excluded. The area of questionable application is usually between 400 Hz and 3 kHz bandwidth. The transfer of these signals on the data bus will depend heavily upon the loading of the bus in a particular application. The decision must be based on projected future bus needs as well as the current loading. Another class of

signals which in general are not suitable for multiplexing are those which can be typified by a low rate (over a mission) but possessing a high priority or urgency. Examples of such signals might be a nuclear event detector output or a missile launch alarm from a warning receiver. Such signals are usually better left hardwired, but they may be accommodated by the multiplex system if a direct connection to the bus controller's interrupt hardware is used to trigger a software action in response to the signal.

10.4 High Reliability Requirements. The use of simple parity for error detection within the multiplex bus system was dictated by a compromise between the need for reliable data transmission, system overhead, and remote terminal simplicity. Theoretical and empirical evidence indicates that an undetected bit error rate of 10^{-12} can be expected from a practical multiplex system built to this standard. If a particular signal requires a bit error rate which is better than that provided by the parity checking, then it is incumbent upon the system designer to provide the reliability within the constraints of the standard or to not include this signal within the multiplex bus system. A possible approach in this case would be to have the signal source and sink provide appropriate error detection and correction encoding/decoding and employ extra data words to transfer the information. Another approach would be to partition the message, transmit a portion at a time, and then verify (by interrogation) the proper transfer of each segment.

10.5 Stubbing. Stubbing is a method wherein a separate line is connected between the primary data bus line and a remote terminal. The direct connection of a stub line causes a mismatch which appears on the waveforms. This mismatch can be reduced by filtering at the receiver and by using Biphase modulation. Stubs are often employed not only as a convenience in bus layout but as a means of coupling a unit to the line in such a manner that a fault on the stub or terminal will not greatly affect the transmission line operation. In this case, a network is employed somewhere in the stub line to provide isolation from the fault. These networks are also used for stubs that are of such length that the mismatch and reflection degrades bus operation. Of the possible networks, transformer coupling gives the least loss. For a 300 foot transmission line operating at 1 megabit, the total loss for the system with 30 stubs was between 19 and 25 dB. For other networks such as direct, loss, and loaded, the loss varied from 32 to 45 dB. If the length of the stub does not approach one-quarter wavelength or is less than 50 cm, it does not seem to cause significant distortion. It may be well to note that stubbing is not a preferred method of bus configuration, but that it is necessary or convenient in the physical layout or installation of the transmission line. The exact limit to stubbing depends upon the number of stubs, length, type of modulation, and the degree of filtering used.

10.6 Status Code Usage. The nine bits in the remote terminal status word allocated for Status Codes may be utilized in any manner the terminal designer wishes. Such usage may include the following possibilities:

10.6.1 Vectored Service Request. These nine software interpretable bits may be used to encode a subaddress and word count referencing a specific set of data words which the terminal wishes to have collected by the bus controller. All zeros in the field could signify no requests.

10.6.2 Error Code Supplement. The nine bits could be used in a manner similar to that discussed in 10.6.1, but rather providing references to subaddressed data words relating to message errors or terminal flags. These referenced words might then contain more detailed information such as parity errors, word count errors, encoding errors, power supply failures, interfaces subsystem failure, etc.

10.6.3 Assigned Codes. The individual bit positions may be assigned specific error code significance. Thus, if one bit is set, then a power supply failure is indicated, or if another bit is set, then a Manchester encoding error has occurred.

ANNEX C

PROVISIONAL RECOMMENDATIONS FOR A SYSTEM OF

MULTIPLEX TRANSMISSIONS VIA BUS LINES

IN AIRCRAFT

CONTENTS

	Page
1. ARCHITECTURE	C-5
1.1 Definition	C-5
1.1.1 Bus	C-5
1.1.2 Unit	C-5
1.1.3 Couplers	C-5
1.2 Reliability	C-6
1.3 Safety	C-6
1.3.1 Safety of transmission	C-6
1.3.2 Integrity	C-6
1.4 Power Supply	C-6
2. STRUCTURE OF THE BUS	C-6
2.1 Type of Structure	C-6
2.2 Method of Connection	C-6
2.3 Types of Connecting Remote Terminals with the Bus Controller	C-7
2.4 Number of Remote Terminals	C-7
3. TYPE OF INTER-EQUIPMENT CONNECTIONS	C-7
3.1 Components of the Transmission Medium	C-7
3.2 Electrical Characteristics of the Transmission Medium	C-7
3.2.1 Insertion loss	C-7
3.2.2 Impedance	C-8
3.2.3 Band pass	C-8
3.2.4 Electrical isolation	C-8
3.3 Passive Shunt	C-8
3.4 Regenerating Shunting Devices	C-8
3.5 Line Matching	C-8
3.6 Screening	C-8
4. CHARACTERISTICS OF THE TRANSMISSION SIGNALS	C-8
4.1 Bit Rate	C-8
4.2 Modulation and Type of Coding	C-8
5. TYPES OF CONNECTIONS	C-9
5.1 Send	C-9
5.1.1 Electrical galvanic isolation	C-9
5.1.2 Short-circuit protection	C-9
5.1.3 Levels	C-9
5.1.4 Waveform	C-9
5.1.5 Noise	C-9
5.2 Receive	C-9
5.2.1 Electrical (galvanic) isolation	C-9
5.2.2 Short-circuit protection	C-9
5.2.3 Detection levels	C-9
5.2.4 Waveform	C-10
5.2.5 Input impedance	C-10
5.2.6 Noise immunity	C-10
5.2.7 Common mode rejection	C-10
6. CHARACTERISTICS OF THE EXCHANGES ON THE BUS LINE	C-10
6.1 Definitions	C-10
6.1.1 Character	C-10
6.1.2 Word	C-10
6.1.3 Message	C-10
6.1.4 Exchange	C-10
6.2 Principle of the Exchange	C-11
6.2.1 Initiation of the Exchanges	C-11
6.2.2 Origin and Destination of the Messages	C-11
6.2.3 Types of Addressing	C-11
6.3 Synchronization	C-11
6.3.1 Nature of the basic clock	C-11
6.3.2 Response time of a peripheral	C-11
6.3.3 Synchronization of words or messages	C-11

	Page
6.3.4 Character synchronization	C-12
6.3.5 Character or word identification	C-12
6.4 Formats	C-12
6.4.1 Format of characters	C-12
6.4.2 Format of messages	C-12
6.5 Structures	C-12
6.5.1 Character structure	C-12
6.5.2 Message structure	C-12
7. (Reserved)	C-12
8. SUB-SYSTEM/BUS COUPLER JUNCTION	C-12
8.1 General	C-12
8.2 Definition of Signals Giving Access to the Sub-Systems	C-13
8.2.1 Time base signals	C-13
8.2.2 Addressing and function signals	C-13
8.2.3 Information signals	C-13
8.2.4 Dialogue signals	C-13
8.2.5 Status signals	C-13
8.3 Definition of the Wires of the Bus Coupler Output Interface	C-13
8.3.1 Time base signals	C-13
8.3.2 Addressing and function signals	C-13
8.3.3 Input information signals	C-13
8.3.4 Dialogue signals	C-14
8.3.5 Status signals	C-14
8.4 Definition of the Wires of the Bus Coupler Input Interface	C-14
8.4.1 Time base signals	C-14
8.4.2 Addressing and function signals	C-14
8.4.3 Output information signals	C-14
8.4.4 Dialogue signals	C-14
8.5 Physical Characteristics of the Connections	C-14
8.5.1 Coding of signals transmitted	C-14
8.5.2 Electrical characteristics of the signals transmitted	C-14
8.5.3 Nature of the connections between sub-system and bus coupler	C-14
9. CONTROL UNIT	C-15
9.1 Types of Exchange Provided	C-15
9.1.1 Bus controller to remote terminals	C-15
9.1.2 Remote terminals to bus controller	C-15
9.1.3 Remote terminal to remote terminal	C-15
9.2 Functions to be Performed	C-15
9.2.1 Organization of exchanges	C-15
9.2.2 Initialization of exchanges	C-15
9.2.3 Monitoring of exchange	C-15
9.2.4 Fault evaluation	C-16
9.2.5 Action required in the event of a persistent fault	C-16
9.2.6 Information about the state of functioning of the system	C-16
9.3 Safety	C-16
9.4 Programme Interrupt	C-16

FIGURES	C-17
---------	------

PROVISIONAL RECOMMENDATIONS FOR A SYSTEM OF MULTIPLEX TRANSMISSIONS VIA BUS LINES IN AIRCRAFT

1. ARCHITECTURE

1.1 Definition (see Figure 1)

1.1.1 Bus

The medium for the transmission of data from a multiple access multiplex system. The bus may consist of one or more lines.

1.1.2 Unit

Name given to any assembly connected to the line or lines of the bus.

1.1.2.1 Control unit(s) (bus controller(s))

Unit(s) controlling the bus.

1.1.2.2 Peripheral unit (remote terminal)

Any unit connected to the line other than the control unit.

1.1.3 Couplers

Name given to functional units forming part of the peripheral units (remote terminals), or of the control unit (bus controller) and integrated or otherwise with these units. Couplers connect the sub-systems to the bus.

1.1.3.1 Bus coupler (Fr. initials: CDB)

The bus couplers (CDB's) perform all the functions connected with transmission by bus lines, which are common to all the units connected to the same transmission by means of bus lines.

Each bus coupler (CDB) will include the following functions:

- send/receive;
- modulation/demodulation;
- bit synchronization;
- synchronization of exchanges (see para. 8.2);
- recognition of type of address (label or physical),
- recognition of the peripheral address (physical addressing);
- the peripheral address is supplied to the bus coupler (CDB);
- generation and validity check of the transmission (e.g. code validity, format, parity);
- formatting and synchronization of the send sequence (e.g. parity generation);
- character serialisation/deserialisation;
- recognition of type of word (information, data procedure).

It could also include the following:

- serial and other (not serial e.g. parallel) information generation;
- recognition of label addresses;
- storing of the whole or part of the message before retransmission.

1.1.3.2 Sub-system coupler (CSS)

The sub-system couplers CSSs, if provided, perform all the dialogue functions appropriate to the sub-systems connected to the bus coupler CDB via the sub-system coupler.

1.1.3.3 Installation of the couplers

The inclusion or otherwise of the couplers (CDB and CSS) in this equipment will depend on the size of the equipment or sub-system.

For very simple equipments the data will for example be regrouped and/or distributed by a sub-system coupler.

In the case of more complicated equipments which in themselves justify a bus coupler (CDB) and a sub-system coupler (CSS), the latter will be readapted.

1.2 Reliability

To be settled later.

1.3 Safety

1.3.1 Safety of Transmission

1.3.1.1 Bit

(See para. 5.2.6.)

1.3.1.2 Messages

To be decided.

1.3.1.3 Exchanges

To be decided.

1.3.2 Integrity

Comment:

Integrity can be provided at the software (procedures) and/or the hardware (redundancy) level.

It will depend on the operational requirements for the systems or sub-systems.

1.4 Power Supply

The couplers may receive their power supply:

- from the sub-system,
- from the incorporated power supply unit.

2. STRUCTURE OF THE BUS

2.1 Type of Structure

Z1: parallel structure (shunt)

Z2: looped series structure (see Figure 2)

Note:

For a bus of approximately 100 metres and for a limited number of peripheral units (remote terminals <32), it would be preferable to choose the parallel structure.

The series structure will be reserved for connections of a more special type which do not meet these criteria.

2.2 Method of Connection

Series structure:

Not applicable.

Parallel structure:

- Z1: short shunt tee-tap (50 cm)
- Z2: long shunt

Note:

The short shunt will preferably be selected. The long shunt will be used only as an exceptional case, since tests will have to be made to determine the maximum permissible value.

2.3 Types of Connecting Remote Terminals with the Bus Controller (bus architecture) (see Figure 3)

- Z1: Structure with 1 two-way bus line (bidirectional)
- Z2: Structure with 2 one-way bus lines
- Z3: Structure with 2 bus lines — 1 two-way and 1 one-way bus line.

2.4 Number of Remote Terminals

- Z: The maximum number of remote terminals which can be connected to a bus shall be at least 16.

Note:

Each transmitter of any unit shall be able to supply at least 16 units which are connected to the bus.

3. TYPE OF INTER-EQUIPMENT CONNECTIONS

3.1 Components of the Transmission Medium

- Z1: Insulated screened twisted pair
- Z2: Coaxial pair
- Z3: Twin coaxial pair

Note 1:

The coaxial pair and the twin coaxial pair will be used only in specific applications.

Note 2:

Definitions:

- Cable: a combination of one or more wires.
- Lines: part of the conductors of a cable or of a combination of cables actually used to transmit an electrical signal between two points.
- Pair: a two-conductor cable.
- Screen: metal covering of a cable which may, or may not, be connected to a floating potential and the purpose of which is to reduce the transfer impedance of the cable sheath.
- Screening: metal covering of a cable which is sufficiently conducting so that each of its points can be considered as being at the same potential, generally earth potential or a reference potential with reference to earth.
- Armouring: metal covering of a cable the sole purpose of which is the mechanical protection of the cable:
- Symmetrical pair: a cable with two conductors of identical structure, arranged in parallel or twisted.
- Coaxial pair: cable with two cylindrical conductors having the same axis.

3.2 Electrical Characteristics of the Transmission Medium

3.2.1 Insertion Loss

- Z1: For an insulated screened twisted pair at 1 MHz, the insertion loss will be less than or equal to:
 - 10 dB/100 m for $L < 30$ m
 - 3 dB/100 m for $30 \text{ m} < L < 100$ m

Note:

For an insulated screened twisted pair the transmission insertion loss is therefore limited to 3 dB.

For a coaxial pair or a twin coaxial pair the insertion loss will be indicated later.

3.2.2 Impedance

Z1: For an insulated screened twisted pair, the characteristic impedance of the cable will be $75 \text{ ohm} \pm 10 \text{ ohms}$.
Distributed capacitance; less than 150 pF/m.

Note:

Values for the coaxial cables to be indicated later.

3.2.3 Band Pass

To be decided.

3.2.4 Electrical Isolation

3.3 Passive Shunt (tee-tap)

To be decided.

Note:

Insufficient information available on which to base a recommendation, particularly in regard to the location of the transformer (electrical isolation).

3.4 Regenerating Shunting Devices

To be decided.

3.5 Line Matching

Z: line matched at both ends.

3.6 Screening

To be decided.

Note:

Some experimental work will be required in order to arrive at a definite recommendation.

4. CHARACTERISTICS OF THE TRANSMISSION SIGNALS

4.1 Bit Rate

Z1: For an insulated screened twisted pair bit rate = 1 Mbits/s

Note:

This value appears to be the best compromise at the present time between the requirements and the technological capabilities.

4.2 Modulation and Type of Coding (see Figure 4)

Z1: Bi-phase code, MANCHESTER, two-level modulation

Z2: Bi-phase code, resetting, 3-level modulation

Z3: Bipolar split phase code (BOEING-LITTON), 3-level modulation.

Note 1:

On the assumption of a connection with electrical isolation by a transformer (see para. 5.1.1), it is necessary to have a zero continuous (dc) component. The three codes meet this requirement.

5. TYPES OF CONNECTIONS

5.1 Send

5.1.1 Electrical Galvanic Isolation

Z: Electrical (galvanic) isolation by means of a transformer.

Note:

The use of an optoelectronic coupler could be considered for future use.

5.1.2 Short-Circuit Protection

Z1: Protection giving a maximum line insertion loss of 2.5 dB in the event of a short circuit in the transmitter

Z2: No protection.

5.1.3 Levels (see Figure 5)

Z1: $\pm 3 \text{ V} \pm 0.3 \text{ V}$ with protective network

Z2: $\pm 6 \text{ V} \pm 1 \text{ V}$ without protective network

Note:

These recommendations are the result of a compromise between different parameters (noise immunity, transmission of unwanted signals, power dissipated, production of the circuits, insertion loss on the line).

5.1.4 Waveform

Z: Rise time (10% to 90%) to be specified. Distortion (exceeding, surging) less than 10% of the nominal value of the signal.

Note:

The rise time is very closely related to the assumed radiation and the type of cable used.

5.1.5 Noise (specific to the transmitter)

To be decided.

Provisional recommendation:

If the terminal is receiving or if it is not being supplied with power, the noise on the line must not be more than 10 mV peak to peak.

5.2 Receive

5.2.1 Electrical (Galvanic) Isolation

Z: Electrical isolation by means of a transformer.

Note:

(See para. 5.1.1.)

5.2.2 Short-Circuit Protection

Z1: Protection giving a maximum insertion loss on the line of 2.5 dB in the event of a short circuit in the receiver.

Z2: No protection.

5.2.3 Detection Levels (see Figure 5)

Z1: Detection levels $+0.6 \text{ V} (\pm 0.1 \text{ V})$ and $-0.6 \text{ V} (\pm 0.1 \text{ V})$ compatible with Z1 in para. 5.1.3.

Z2: Detection levels $+2.25 \text{ V} (\pm 0.25 \text{ V})$ and $-2.25 \text{ V} (\pm 0.25 \text{ V})$ compatible with Z2 in para. 5.1.3.

Note:

A firm recommendation is to be based on experimental results.

5.2.4 Waveform

To be specified.

Note:

Lack of information prevents formulation of a recommendation. The waveform extending from the square wave to the sine wave is valid only for the MANCHESTER bi-phase code.

5.2.5 Input Impedance

To be specified.

Note:

A recommendation on this point can only be based on experimental results.

5.2.6 Noise Immunity

To be specified.

Note:

A recommendation on this point can only be based on experimental results in order to define an error rate per bit and a signal-to-noise ratio.

5.2.7 Common Mode Rejection

To be specified.

Provisional recommendation

In the frequency band 0 to 2 MHz any signal with a peak amplitude less than or equal to ± 25 V between the line and earth must not cause any deterioration in the performance of the terminal.

Any signal with a peak amplitude of ± 50 V or less must not cause any permanent damage to the terminal.

6. CHARACTERISTICS OF THE EXCHANGES ON THE BUS LINE

6.1 Definitions

6.1.1 Character

An element made up of a fixed number of bits which are transmitted as an entity.

6.1.2 Word

An information element made up of a number of characters constituting an entity from the "information" point of view.

6.1.3 Message

A collection of words in transit on the bus as a single block and in a single direction.

Example:

An instruction message transmitted by the control unit (bus controller), an information message transmitted by a peripheral remote terminal, status message etc.

6.1.4 Exchange

A collection of messages in transit on the bus from a single initiative taken by the unit controlling the bus.

Example:

Sending of data and acknowledgement transmitted by the receiver.

Note:

Each character, word, message or exchange may comprise check elements (e.g. parities) and these are then included in the above definitions.

6.2 Principle of the Exchange (message routing)

6.2.1 Initiation of the Exchanges

Z: Exchanges are initiated by the unit (bus controller) controlling the bus.

6.2.2 Origin and Destination of the Messages

Z1: Control unit (bus controller); Peripheral unit (remote terminal);
Peripheral unit (remote terminal); Control unit (bus controller);
Peripheral unit n (remote terminal); Peripheral unit m (remote terminal).

Z2: Control unit (bus controller); n peripheral units (remote terminals);
Peripheral unit (remote terminal); Control unit (bus controller);
Peripheral unit (remote terminal); n peripheral units m (remote terminal) except with a one-way connection.

Note:

Dialogue between peripheral units (remote terminals) is possible only with a two-way connection.

6.2.3 Types of Addressing

Z1: Addressing by physical unit
Z2: Physical addressing and label addressing.

6.3 Synchronization

6.3.1 Nature of the Basic Clock

Send:

Z1: Local clocks

Z2: Clock extracted from the bit synchronization.

Receive:

Z: Clock extracted from the bit synchronization.

Note:

The recommendation Z1 (local clocks) is valid for all the structures defined in para. 2.3.

The recommendation Z2 (clock extracted from the bit synchronization) is valid only for the solution using 2 lines and also for the series structure (see Z2 and Z3 in para. 2.3).

6.3.2 Response Time of a Peripheral

Z: Delay 20 μ s following the last bit in the instruction word or message.

Note:

There are in fact two cases to be considered:

- (i) A peripheral (remote terminal) in which it is certain that the data item is available (or that it can be acquired very quickly at the time of the interrupt):
 - transmission via a one-way line: delay limited to 20 μ s.
- (ii) A peripheral (remote terminal) in which data arrive at random instants or are the result of very long calculation times:
 - either an instruction to prepare the data;
 - or an interrogation procedure (polling) to find out whether the data are available.

The interrogation word or message (polling) is a special case of the instruction word or message in which the data transmitted relate to the status of the peripheral.

6.3.3 Synchronization of Words or Messages

Z1: Synchronization by invalid code
Z2: Synchronization by procedure bits
Z3: Synchronization by envelope detection.

Note:

The synchronization mode is dependent on the type of modulation, the structure of the messages and the detection methods. It implies fairly complex circuitry and provides a more or less effective check of the data transmitted.

6.3.4 Character Synchronization

- Z1: Synchronization by counting from the word or message synchronization
- Z2: Synchronization by invalid code
- Z3: Synchronization by envelope detection.

6.3.5 Character or Word Identification

- Z1: Identification by synchronization
- Z2: Identification by position in the message
- Z3: Identification by bits of the character.

Note:

If a particular type of synchronization is carried out, this may be used to differentiate types of characters or words; otherwise an operational code is used, made up of a few bits of the character, and the word or character is identified by its position in the message.

6.4 Formats**6.4.1 Format of Characters**

- Z: A single type of format consisting of 1 octet (8 bits of information not including synchronization or check bits).

6.4.2 Format of Messages

To be specified.

Note:

Recommendations will be issued later, based on the results of current studies.

6.5 Structures**6.5.1 Character Structure**

To be specified.

6.5.2 Message Structure

Every message is composed of words of the following two types:

- (a) procedure words (instruction, addressing, check ...) the structure of which must be permanently fixed in relation to a particular application;
- (b) data words whose structure is relatively flexible within the same application.

Apart from this sharp distinction between two families of words, it is not *a priori* possible to lay down the various meanings of the procedure words. It may even be felt, with some justification, that these would vary according to the applications.

7. (Reserved)**8. SUB-SYSTEM/BUS COUPLER JUNCTION****8.1 General***Purpose of the Recommendations*

The purpose of these recommendations is to standardize the connection of the sub-systems to the bus coupler. This is without prejudice to the use or otherwise of a sub-system coupler.

The recommendations permit optimization of the bus coupler/bus-system coupler unit (e.g. avoids duplication of buffer stores).

Note:

The similarity of the input and output signals of the sub-system/bus coupler is due to a definition of the wires in the interface, which is still too wide.

Only the functions of the interface wires have been defined at the present time. Physically, some of these may be omitted or regrouped (multifunction, two-way).

8.2 Definition of Signals Giving Access to the Sub-Systems

A sub-system is accessed by means of five types of signals. These signals are defined from the point of view of their function, without prejudice to their use, omission or combination.

8.2.1 Time Base Signals

Signals for organizing exchanges in time.

8.2.2 Addressing and Function Signals

Signals characterizing a type of exchange, as defined by the BUS procedure.

8.2.3 Information Signals

Signals representing the information element (data ...) to be exchanged between the bus coupler and the sub-system coupler.

8.2.4 Dialogue Signals

Signals for transferring information elements.

8.2.5 Status Signals

These are signals which ensure the correct functioning both of the transmission and of the connected sub-system.

8.3 Definition of the Wires of the Bus Coupler Output Interface

8.3.1 Time Base Signals

8.3.1.1 Clock signals

- bit clock wire
- character clock wire
- word clock wire.

8.3.1.2 Phase signals

Note:

These signals, to be defined in relation to the bus procedure, make it possible to distinguish the various phases in the exchanges.

8.3.2 Addressing and Function Signals

Note:

To be defined in relation to the bus procedure adopted.

8.3.3 Input Information Signals

The information elements sent out by the bus coupler are transmitted along these wires:

- Z1: in series (bit by bit)
- Z2: in parallel.

8.3.4 Dialogue Signals

Note:

To be defined in relation to the bus procedure adopted.

8.3.5 Status Signals

Note:

To be defined according to the bus procedure adopted.

8.4 Definition of the Wires of the Bus Coupler Input Interface

8.4.1 Time Base Signals

8.4.1.1 Clock signals

- bit clock wire
- character clock wire
- word clock wire.

8.4.1.2 Phase signals

Note:

These signals, to be defined according to the bus procedure adopted, make it possible to distinguish the various phases in the exchanges.

8.4.2 Addressing and Function Signals

Note:

To be defined according to the bus procedure adopted.

8.4.3 Output Information Signals

The information elements sent out by the sub-system coupler are transmitted along these wires:

- Z1: in series (bit by bit)
- Z2: in parallel.

8.4.4 Dialogue Signals

Note:

To be defined according to the bus procedure adopted.

8.5 Physical Characteristics of the Connections (interfacing signals)

8.5.1 Coding of Signals Transmitted

- Z: Non-return to zero.

Note:

In some cases this "non-return to zero" signal will be maintained only during the period of time of the "information present" signals.

8.5.2 Electrical Characteristics of the Signals Transmitted

Note:

This item depends on the choice of technology for the bus coupler and it is impossible to make any recommendation in this respect.

8.5.3 Nature of the Connections between Sub-System and Bus Coupler

8.5.3.1 Bus coupler and sub-system connected mechanically or integrated

- Z: Single wire cabling.

8.5.3.2 Bus coupler and sub-system mechanically separate

Z: Screened two-wire line used in the differential mode.

9. CONTROL UNIT (bus controller, Fr. initials: UG)

A control unit (bus controller) controls (or manages) the exchange of data on the bus.

It can be:

- a specialized equipment which performs this function only;
- one of the components of the system (flight control computer, navigational computer etc.) which performs both its own functions and the control function.

Several units may act in turn as a control unit (bus controller) within the same system.

The choice will depend on the system. This Section is confined to the guiding principles on which the design has been based.

9.1 Types of Exchange Provided (ref. para. 6.2)

The control unit (bus controller) (UG) originates any exchange between the peripheral units (remote terminals) (Fr. initials: UP) connected by the bus. The exchanges are carried out by means of the following messages:

9.1.1 Bus Controller to Remote Terminals (UG to UP(s))

Transmission of procedures and/or data.

9.1.2 Remote Terminals to Bus Controller (UP to UG)

Reply to a bus controller to remote terminal (UG-to-UP) transmission, which may contain data, an acknowledgment of receipt, or status information.

9.1.3 Remote Terminal to Remote Terminal (UP to UP(s))

Reply to a bus controller to remote terminal (UG-to-UP) transmission, which generally contains data and/or procedures.

9.2 Functions to be Performed

9.2.1 Organization of Exchanges

- either according to a variable programme;
- or according to a fixed programme.

9.2.1.1 Variable programme

The organization of the exchanges must be capable of variation in relation to:

the flight phase, the environment, the system configuration, either by external control (e.g. from the cockpit), or automatically using particular criteria (e.g. under-carriage down).

9.2.1.2 Fixed programme

The exchanges follow a fixed sequence.

9.2.2 Initialization of Exchanges

Any exchange on the bus is initiated by the bus controller (UG).

9.2.3 Monitoring of Exchange

The bus controller (UG) monitors all the exchanges to ensure correct functioning. Monitoring covers several functions, for example:

- (a) checking the quality of the transmission (check bits, code format etc.);
- (b) checking replies, as necessary;
- (c) comparing replies repeated sequentially or in parallel.

9.2.4 Fault Evaluation

The bus controller (UG) must distinguish between transmission faults and failures of the exchange system.

9.2.5 Action Required in the Event of a Persistent Fault

This may consist of automatic reconfiguration of the system, or, if this is impossible, of the display of the relevant data to the crew.

Note:

To be defined in relation to the bus system adopted.

9.2.6 Information About the State of Functioning of the System

This information is essential for reconfiguration of the system. It will be available for display to the crew, either automatically or on demand, and also for recording purposes.

This information may comprise the following elements:

- identification of the devices or sub-systems which have failed;
- nature of the fault;
- present configuration of the system.

9.3 Safety

The bus controller (UG) will incorporate the necessary automatic test and/or checking facilities to ensure that the equipment functions with the degree of integrity compatible with the proposed application.

9.4 Programme Interrupt

In the case of exchanges organized in accordance with a variable programme the bus controller (UG) can be provided with logic circuits for hierarchical interrupts for:

- modifying the organization of the exchanges;
- reconfiguration of the system (with transfer, as necessary, of the main control).

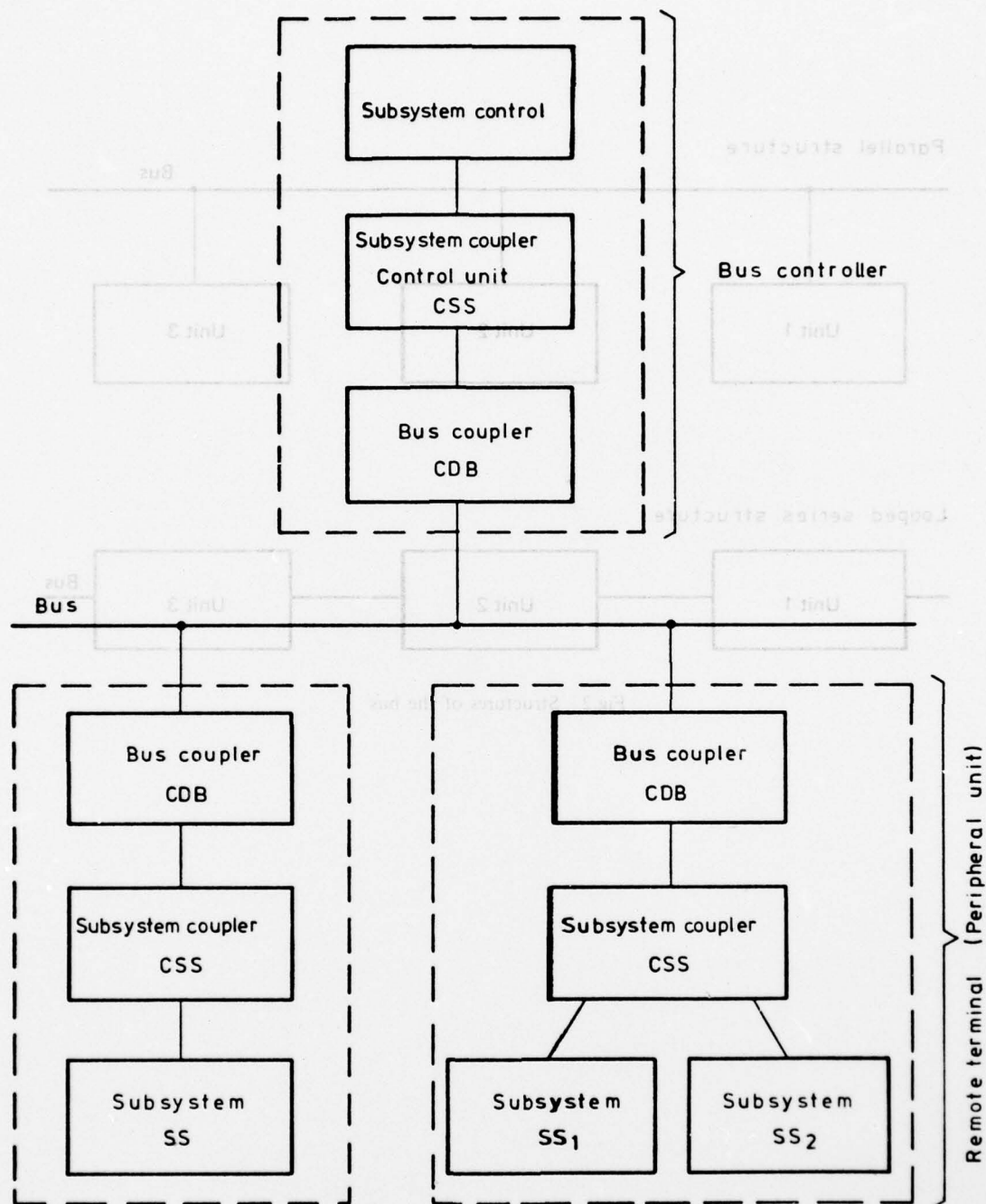
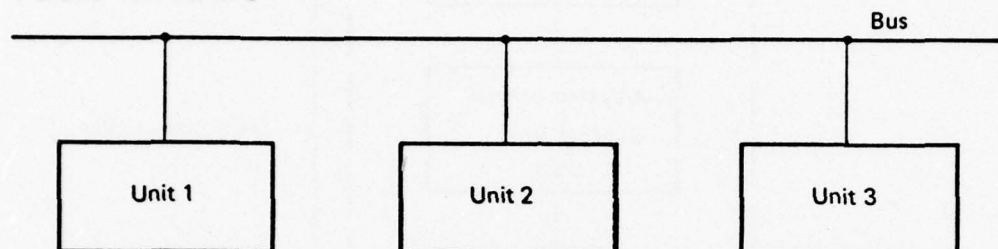


Fig.1 Bus architecture

Parallel structure



Looped series structure

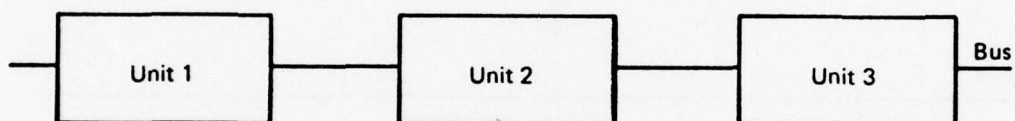
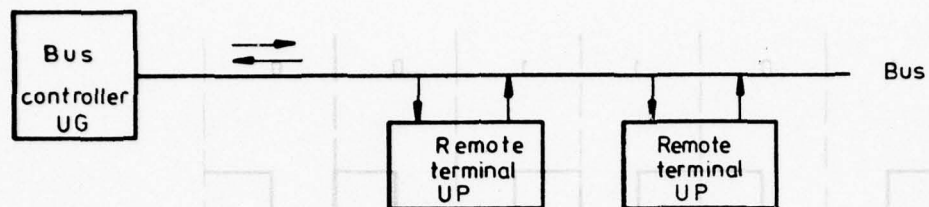
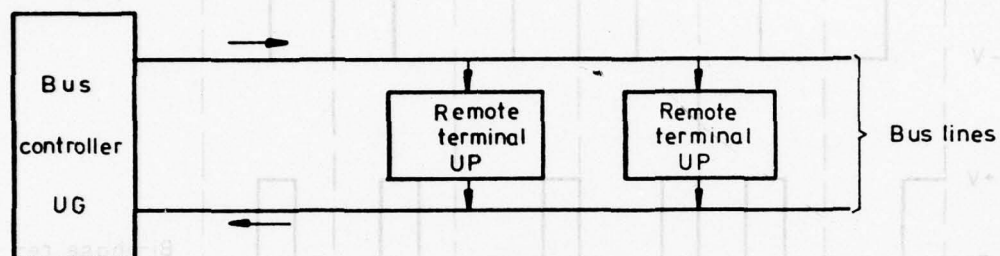


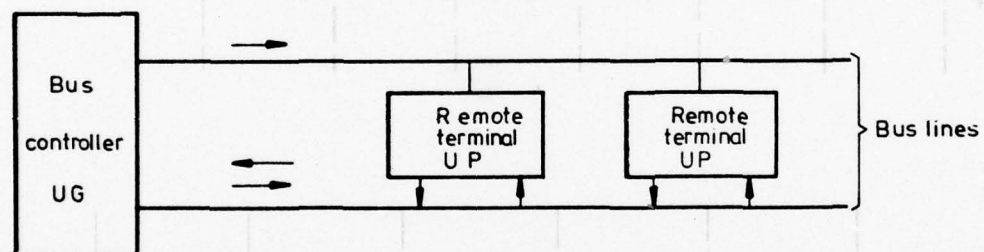
Fig.2 Structures of the bus



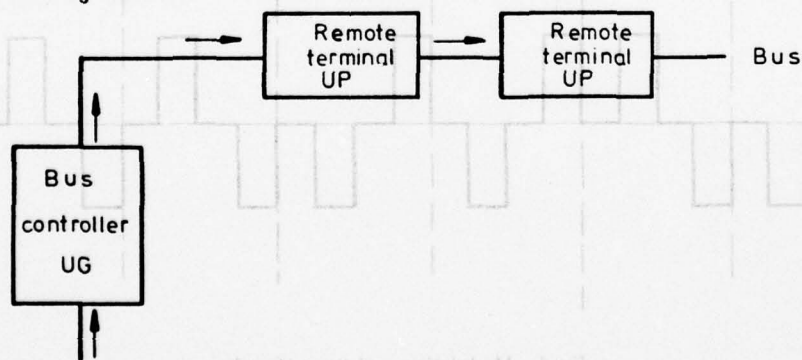
a) Structure with one bidirectional bus line



b) Structure with two unidirectional bus lines



c) Structure with two bus lines, one unidirectional e.g. procedure line, one bidirectional e.g. data line



d) Example for a looped series has structure

Fig.3 Examples of different bus structures

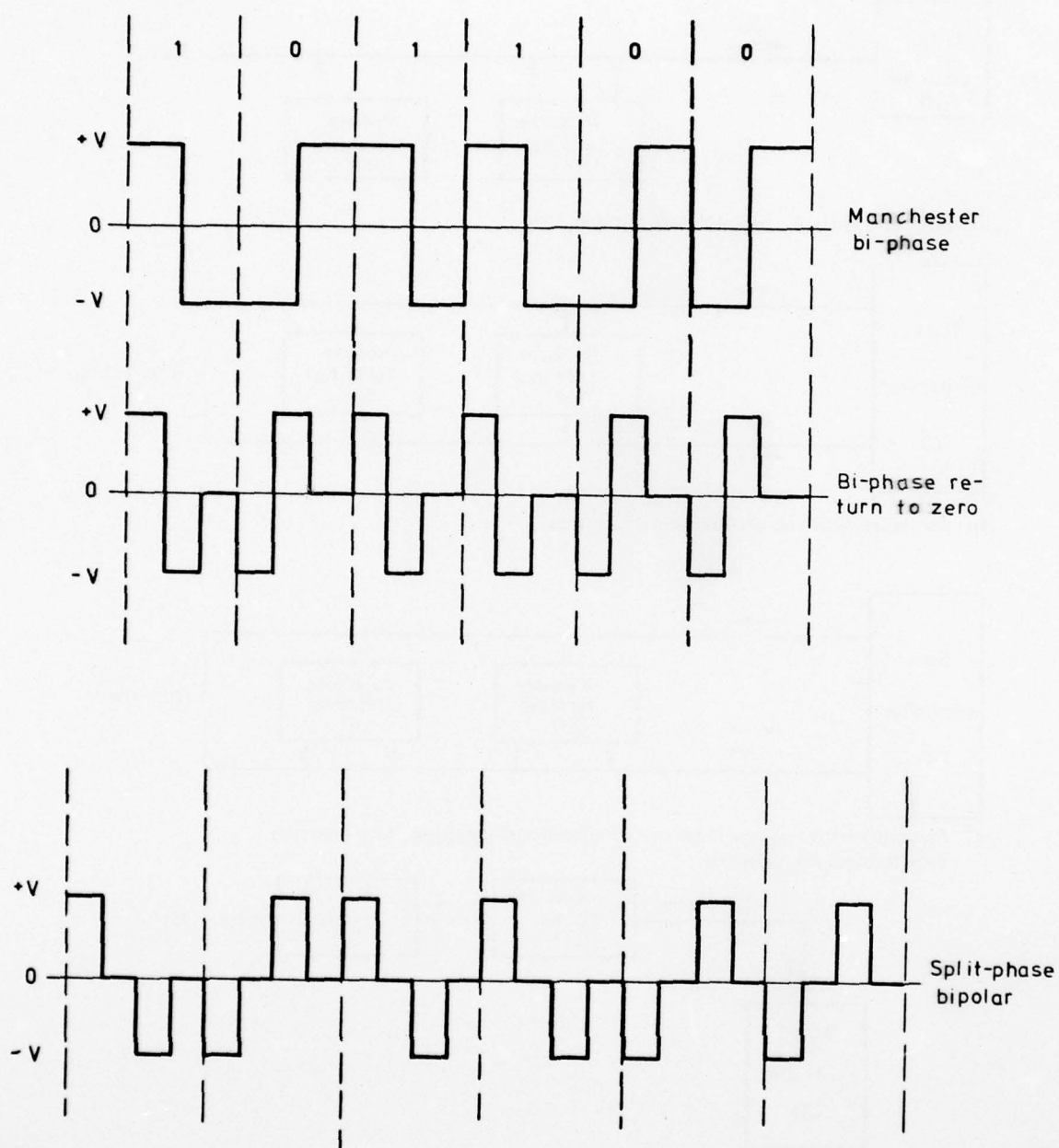


Fig.4 Modulation and type of coding

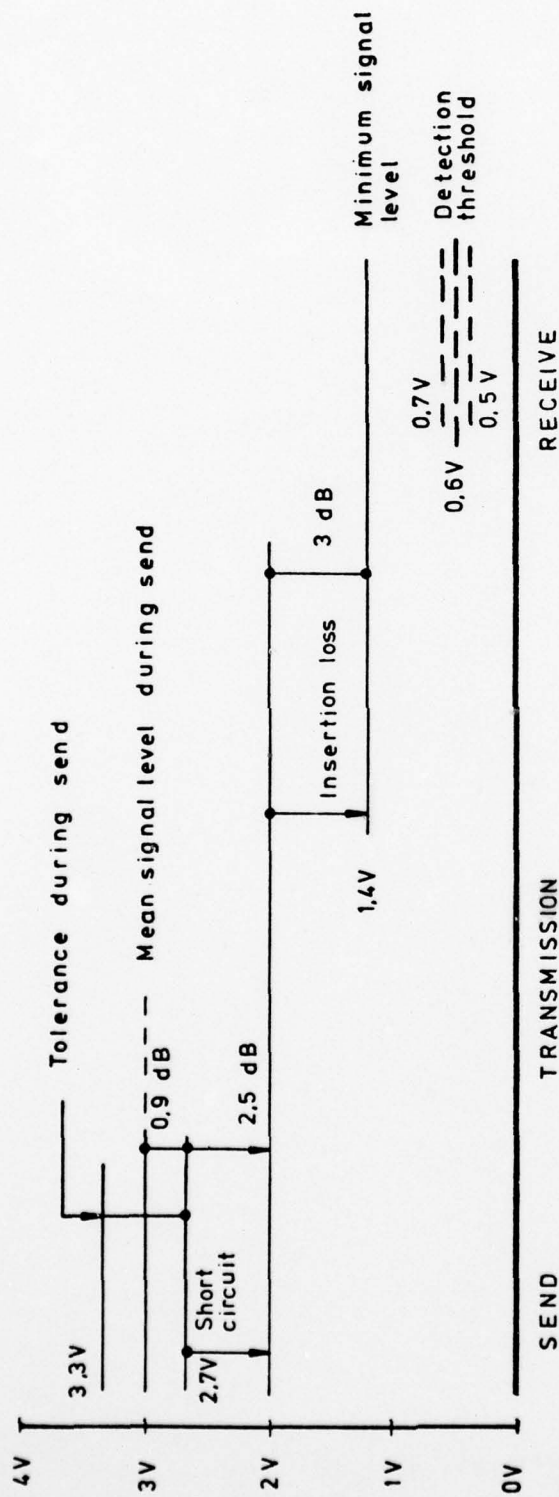


Fig.5 Voltage levels on bus lines for a send level of ± 3 V $\pm 10\%$ and for a receive level of ± 0.6 V ± 0.1 V

ANNEX D

DEFINITION OF THE DIGIBUS FOR

THE EXCHANGE OF DIGITAL DATA

IN A COMBAT AIRCRAFT

CONTENTS

	Page
1. GENERAL	D-5
1.1 Note	D-5
1.2 Definitions	D-5
1.3 Safety	D-5
1.4 Power Supply for the Transmission System	D-6
2. STRUCTURE OF THE TRANSMISSION MEDIUM	D-6
2.1 Type of Structure	D-6
2.2 Method of Connection	D-7
2.3 Type of Connection	D-7
3. TYPE OF INTER-EQUIPMENT CONNECTIONS	D-7
3.1 Constitution of the Transmission Medium	D-7
3.2 Electrical Characteristics of the Line	D-7
3.3 Screening	D-8
3.4 Connection of the Equipment	D-8
4. CHARACTERISTICS OF THE TRANSMISSION SIGNALS	D-10
4.1 Bit Rate	D-10
4.2 Modulation and Type of Coding	D-10
5. TYPE OF CONNECTIONS	D-10
5.1 Transmission	D-10
5.2 Reception	D-12
6. CONTROL OF DATA EXCHANGES	D-12
6.1 Definitions	D-12
6.2 Character Format	D-12
6.3 Control of the Digibus	D-12
6.4 Origin and Destination of the Messages	D-13
6.5 Synchronisation of the Peripherals	D-13
7. STRUCTURE OF THE EXCHANGES	D-14
7.1 Types of Words	D-14
7.2 Types of Addressing	D-14
8. STRUCTURE OF THE MESSAGES	D-15
8.1 Word Structure	D-15
8.2 Structure of the Exchanges	D-18
8.3 Overall (Synoptic) View of the Exchange System	D-19
9. MONITORING OF THE EXCHANGE SYSTEM	D-19
9.1 Monitoring of the Exchanges at the Level of Each Peripheral	D-19
9.2 Monitoring of the Exchanges at the Level of the Main Control Unit	D-19
9.3 Monitoring of the Main Control Unit by the Standby Control Unit	D-20
10. RECONFIGURATION	D-20
10.1 Failure of a Peripheral	D-20
10.2 Failure of a Digibus	D-21
10.3 Failure of a Control Unit	D-21
11. MAINTENANCE PREPARATION	D-21
12. DEFINITION OF THE SUBSYSTEM JUNCTION OF THE BUS COUPLER	D-22
12.1 Definition of the Bus Coupler	D-22
12.2 Definition of the Standard Interface	D-22
12.3 Dialogue Procedures	D-22

AD-A044 915

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 3/1
A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CON--ETC(U)
MAY 77 G E SCHWEIZER, A A CALLAWAY, E C GANGL

AGARD-AR-90

NL

UNCLASSIFIED

2 OF 6
ADA
044 915



LIST OF PLATES

1. Procedures for exchanges via the digibus.
2. Organisation of the exchange system.
3. Procedure used for various response times of the peripherals.
4. Standard bus-coupler (COS): Block diagram and connections.
5. Procedure for exchange of a character between the COS and CSS.
6. Procedure for connecting the CSS to the digibus – Instruction phase.
7. Typical exchange procedure – Chronograms.
8. Typical exchange procedure – Chronograms.
9. Bit positions of the various characters in the receive bus (BR) and the send bus (BE).

(COS = COuplu Standard – Standard coupler

CSS = Couplu de sous-système – Sub-system coupler

BE = Bus émission – Send bus

BR = Bus réception – Receive bus).

Exchange of digital data via the digibus.

1. GENERAL

1.1 Note

The definition of the exchange of digital data via a digibus conforms with the recommendations published by the Technical Committee on Integration (April 1974).

1.2 Definitions

1.2.1 Digibus

The medium for the transmission of digital data in a multiplex system.

1.2.2 Control Unit (bus controller)

The unit which organises the exchanges in the digibus.

1.2.3 Peripheral Unit (remote terminal)

Each peripheral unit connected to the digibus comprises:

- the equipment itself;
- the couplers connecting the equipment to the digibus.

1.2.4 Couplers

Name given to a functional assembly forming part of each peripheral and generally integrated with each of them; the assembly connects the actual equipment to the digibus.

This functional assembly consists of the following:

- (a) *one or more bus couplers* (according to the redundancy provided)

The bus coupler is the component which connects the peripheral to the digibus; its definition (excluding redundancy) and the functions which it performs are identical for all equipments. It is related only to the definition of the transmission system and to the organisation of the data exchanges in the digibus. It performs all the logic operations to be carried out by all the equipments connected to the same bus.

The requirements for the bus coupler interface on the equipment and for the procedure for the exchanges between it and the sub-system coupler are given in Section 12.

- (b) *a sub-system coupler (CSS)*

The sub-system coupler performs the functions associated with the transmission system which are not included in the bus coupler or couplers, and which are therefore specific to the requirements of the equipment (remote terminal) in which it is incorporated.

From the physical point of view the sub-system coupler links the actual peripheral unit to the bus coupler or couplers. (See Figure 1).

1.2.5 Digital Transmission System

The transmission system comprises:

- the digibuses;
- the control units (bus controller);
- the bus couplers for each equipment (remote terminal).

1.3 Safety

To ensure good safety of the data exchanges and to guard against a preliminary failure, of whatever kind, provision must be made *inter alia* for redundancy at the following levels:

- digibus;
- control unit (bus controller);
- bus coupler for each equipment (remote terminal).

The sub-system coupler is considered as a functional part of the equipment (remote terminal) and not of the digital transmission system; it need not therefore be redundant if the remote terminal itself is not redundant.

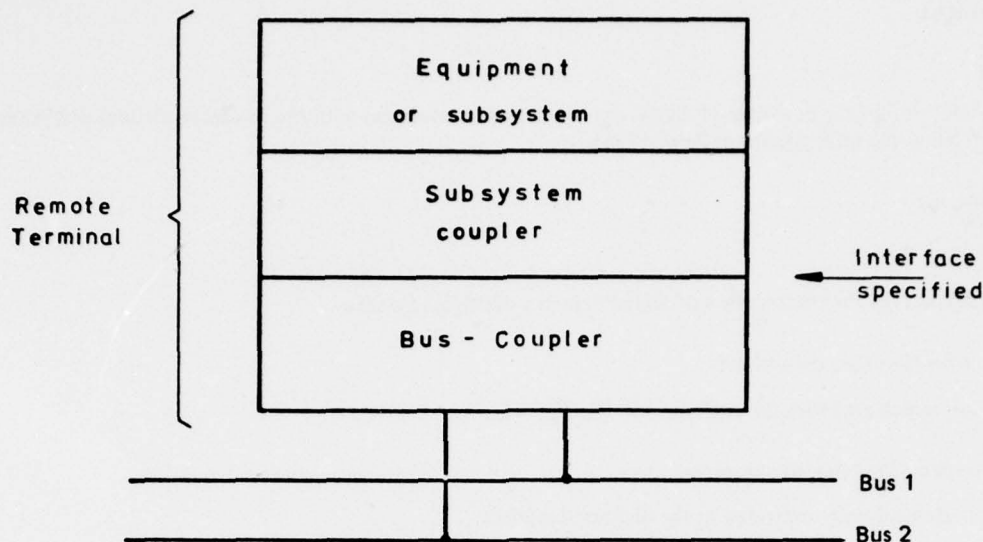


Fig.1 Block diagram of a remote terminal

Main control and emergency control of the buses are provided by two separate units (bus controllers), identical or otherwise, each of which can, if necessary, be included in one of the system computers.

The standby control unit (bus controller) is capable of controlling all the exchanges remaining after failure of the main bus controller.

It is however important to note that each bus controller (main and standby) can control either of the two digibuses (if one of the digibuses fails).

There are several ways of providing redundancy in the bus-coupler of a remote terminal.

This coupler consists of:

- data transmission and reception circuits;
- logic circuits.

Depending on the technology used and the number of remote terminals in the system, redundancy has been specified at various levels, as shown in Figures 2(a) and 2(b).

1.4 Power Supply for Transmission System

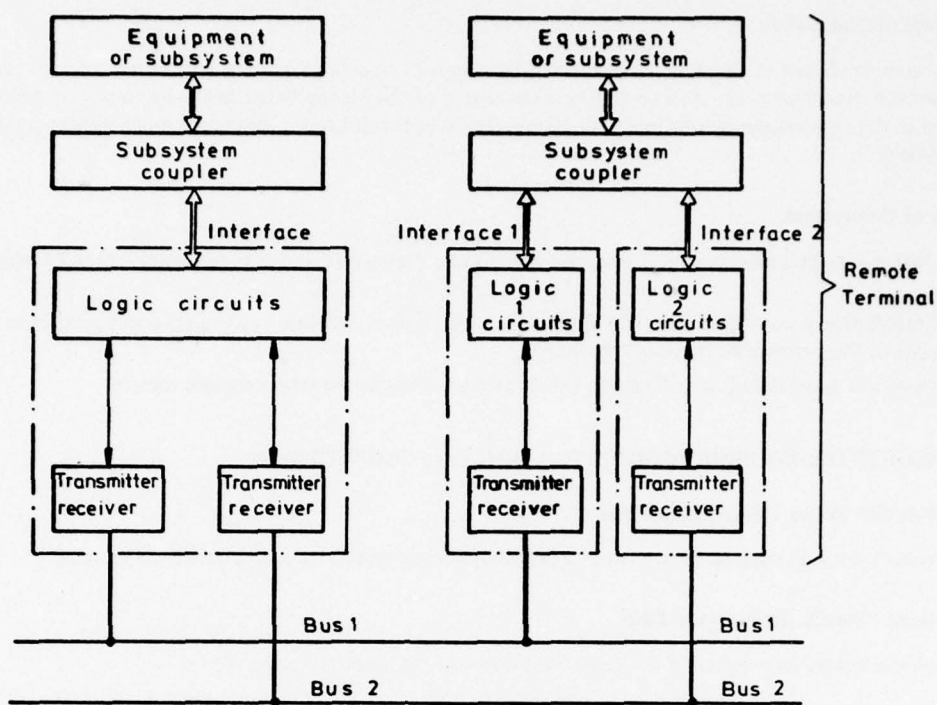
Two sets of conditions may arise for bus-couplers:

1. Bus-coupler specific to an equipment (remote terminal) and incorporated in it (general case). The bus-coupler has no separate power unit and power is supplied by the remote terminal (see Equipment interface specification in Section 1.2)
2. Bus-coupler common to several peripherals
 - *separate bus-coupler*: supplied with power from its own power unit
 - *bus-coupler incorporated in one of the peripherals which it serves*: Power is supplied to it by the peripheral remote terminal in which it is incorporated. (See Figure 3).

2. STRUCTURE OF THE TRANSMISSION MEDIUM

2.1 Type of Structure

The equipments (peripherals or remote terminals P_i) are connected to the digibus in shunt. The basic principle is shown in diagrammatic form in Figure 4.

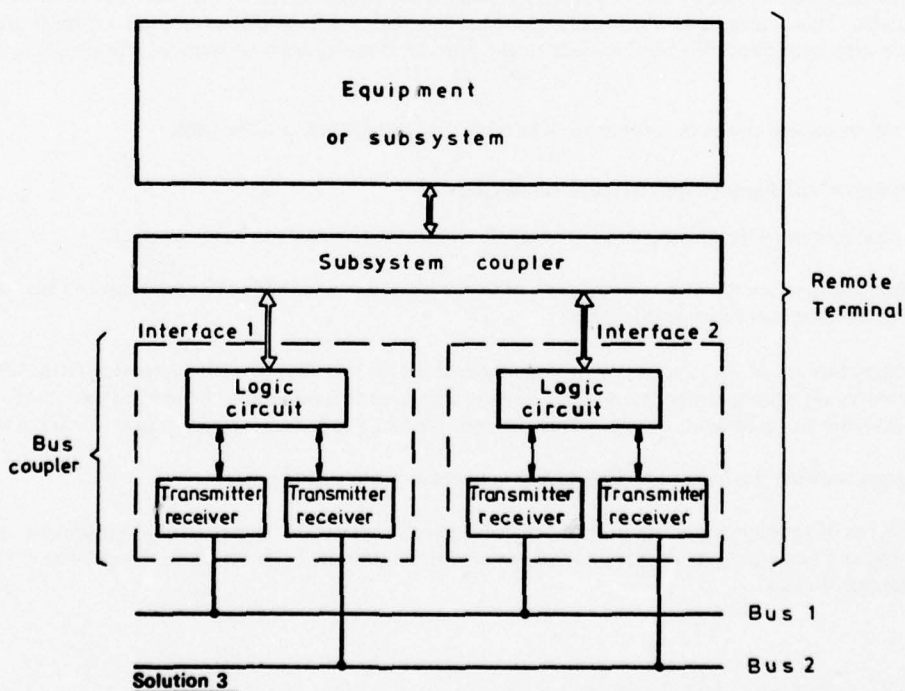
Solution 1

Single redundancy on the logic circuits
and double redundancy on the interfaces
of the busses

Solution 2

Double redundancy on the logic circuits
and on the interfaces to the busses

Fig.2(a) Redundancy in the Remote Terminal at various levels (in solution 2 it should be noted that each bus coupler is specific to a digibus)

Solution 3

Simple redundancy on the logic circuits and double redundancy on the
interfaces to the busses.

Fig2(b) Redundancy in the remote terminal at various levels

2.2 Method of Connection

The equipments and the digibus are connected by means of transformers with a short tee-tap: *the connection is made inside the connector plug*. As a general rule the length of the tee-tap between the beginning of the tee-tap and the connection to the bus-coupler will be less than 30 cm. In exceptional cases it may be as much as 50 cm (subject to agreement by AMD).

2.3 Type of Connection

The digital data exchange system is designed around two digibuses (double redundancy), each of which has two lines:

- a unidirectional procedure line (LP) along which the control unit (bus controller) sends procedure (command) words to the peripherals (remote terminals);
- a two-directional data line (LD) along which data are transmitted (transmit and receive).

3. TYPE OF INTER-EQUIPMENT (REMOTE TERMINAL) CONNECTIONS

3.1 Constitution of the Transmission Medium

The cable used is twisted and screened pair attached at each end to its characteristic impedance.

3.2 Electrical Characteristics of the Line

The characteristic impedance of the cable used is of the order of 75 ohms \pm 5%.

Z_0 will be chosen as near the actual value as possible.

The peripherals (remote terminals) connected to the bus line must have an input impedance which is adequate for connecting 32 subscribers and must be capable of transmitting on an impedance of $\frac{Z_0}{2} = 37.5$ ohms. (Reflexion coefficients and signal power for transmission must be designed to meet these requirements).

3.3 Screening

A preliminary analysis shows that the cable screening must be connected to the chassis earth of the equipments (remote terminals). This connection, which will be as short as possible, will be made from the screening attachment point in the special-to-type connector plug on the inside or the outside of the equipment (remote terminals) (to avoid radiation of noise).

Tests will be carried out in an aircraft to determine the wiring method to be used.

3.4. Connection of the Equipments (Remote Terminals)

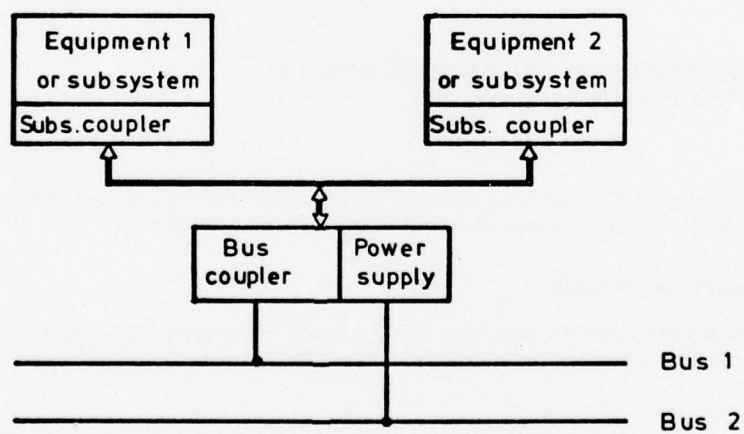
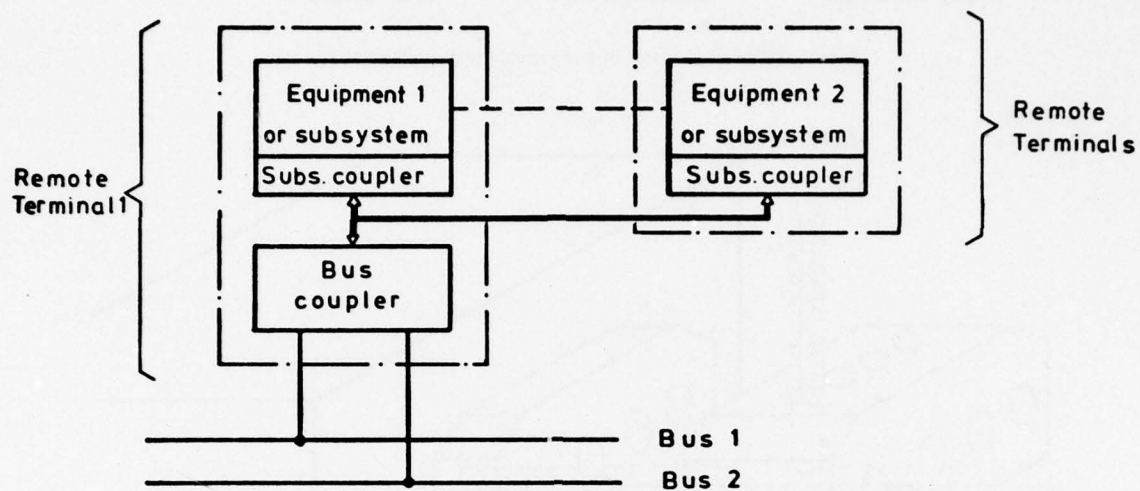
Eight connections (4 lines) connect each equipment (remote terminal) to the two buses.

The connection in tee-tap of a remote terminal to the buses is made inside the connectors on the aircraft. Continuity of the earthing is ensured at this level.

The connectors are of the plug-in type and are located on the rear face of each equipment (remote terminal). This system eliminates any wiring on the front of the units and reduces to a minimum the manual operations required for inserting or removing an equipment (remote terminal) since the latter automatically plugs in to the bus connectors.

A diagram showing the basic principles of this connector is shown in Figure 5.

Other types of special-to-type connectors for the digibus (in particular round ones) are proposed; these do not include a tee-tap and two would be required to connect a remote terminal to the digibus. These connectors may be used only in exceptional cases.



Bus coupler with separate power supply

Fig.3 Power supply of remote terminals

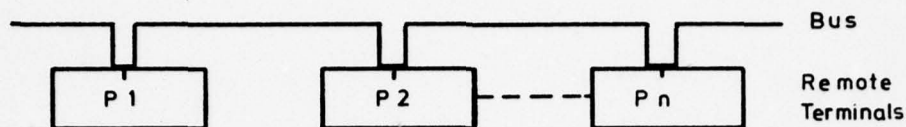


Fig.4 Interconnection of the remote terminals to the bus

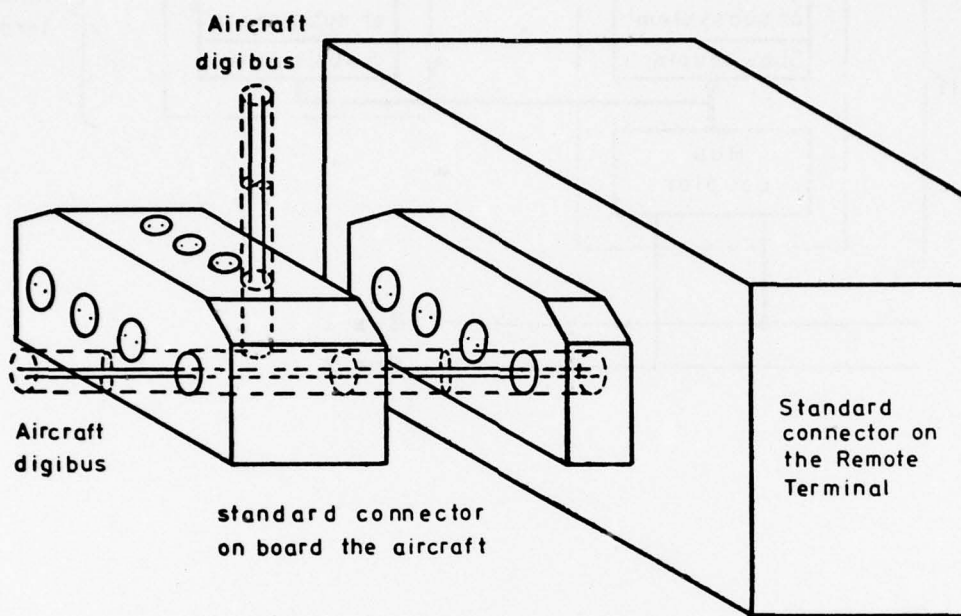


Fig.5 Basic principle of the digibus special-to-type plug-in connectors

4. CHARACTERISTICS OF THE TRANSMISSION SIGNALS (ILLUSTRATIVE PURPOSES ONLY)

4.1 Bit Rate

The bit rate depends on the data exchange requirements and on the extension capabilities required. A rate of 1 Mbits/s has been adopted based on requirements and technological capabilities.

4.2 Modulation and Type of Coding

The code used is a two-phase resetting code with three level modulation. The continuous component (dc signal is zero to permit satisfactory use of the transformer coupling. (See Figure 6).

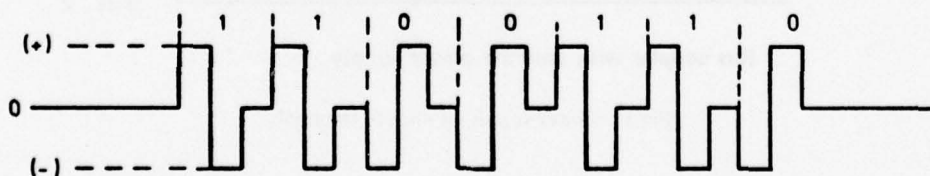


Fig.6 Two phase resetting code with three level modulation [+, -, 0]

5. TYPE OF CONNECTIONS (FOR ILLUSTRATIVE PURPOSES ONLY)

5.1 Transmission

Electrical insulation: achieved by transformer.

Levels: (see Figure 7): $\pm 6V \pm 1V$ without protective network.

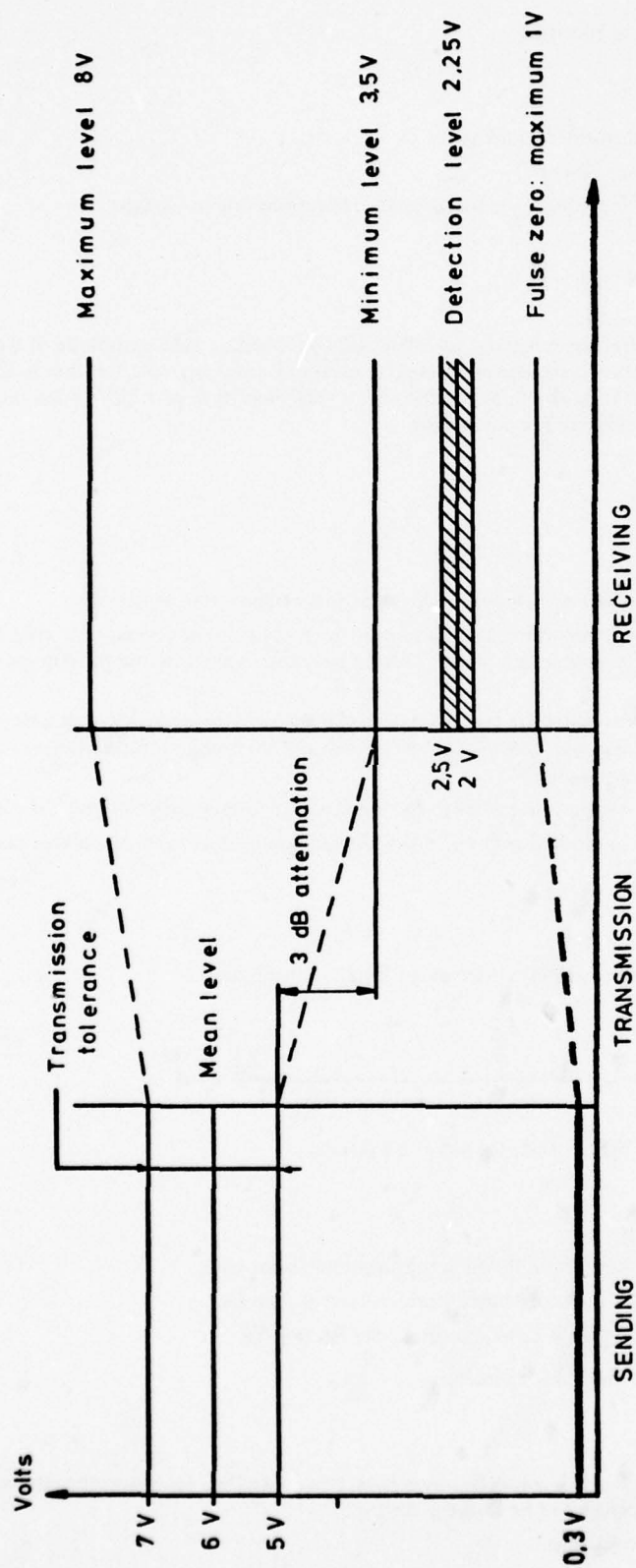


Fig.7 Voltage level* on the bus lines for a transmitting level $\pm 6V$ [$\pm 1V$]

* The thresholds and levels will be fixed more precisely later on, depending on the cables used.

False zero: $\pm 0.3V$ (from $Z_0/2$).

Waveform: *rise time* (10% to 90%): to be specified.

distortion: (exceeding, surging) less than 10% of the nominal value of the signal: (600 mV peak to peak).

Inherent noise in the transmitter: to be specified.

5.2 Reception

Electrical insulation: achieved by transformer coupling

Detection levels: $\pm 2.25V \pm 0.25V$ (see Figure 7)

False zero: $\pm 1V (\pm 0.3V \pm 0.7V)$: disturbances produced by the reflections due to tee-taps

Waveform: to be specified.

Input impedance: to be specified (see 3.2).

Noise immunity: to be specified.

Rejection of common mode: In the frequency band 0 to 2MHz, any signal with a peak amplitude of less than or equal to $\pm 25V$ between the line and the earth must not cause any deterioration in the performance of the equipment. In addition, any signal with a peak amplitude of $\pm 50V$ or less must not cause any permanent damage to the equipment.

6. CONTROL OF DATA EXCHANGES

6.1 Definitions

Character: an information element made up of 10 bits which are transmitted as an entity

Word: a collection of characters which are transmitted as an entity. The format of a word is implicit and fixed during the instruction phase (2 characters) and explicit during the data phase (1 to 4 characters)

Message: a collection of words which are conveyed onto the bus as a single block and in a single direction: e.g. an instruction message transmitted by the bus controller or an information message transmitted by a remote terminal

Exchange: a collection of messages conveyed onto the bus from a single initiative by the bus controller

Data block: a variable number of characters (1 to 256) which are grouped to form data words made up of 1 to 4 characters.

6.2 Character Format

Each character comprises 10 bits grouped as in the following format: (see Figure 8).

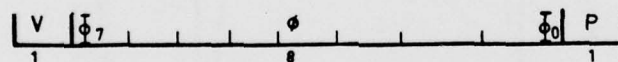


Fig.8 Data format of a character

The bit V may have various meanings:

- the validity bit associated with the data word of which the character forms part;
- procedure linking bit associated with the procedure characters (see Section 8);
- transmission validity bit associated with the echo characters (see Section 8).

The bit P is the odd parity relating to the whole character.

6.3 Control of the Digibus

Control of the exchanges is centralised as it is controlled by a fixed bus controller. However, the data must not pass through the bus controller. Two bus controllers can be used for control:

- Bus controller Unit 1 or main bus controller
- Bus controller Unit 2 or stand-by bus controller.

Each of these two bus controllers can be included in one of the computers* in the system.

- the transmitting remote terminals transmit their data on the data line (LD) to the bus controller alone;
- the receiving remote terminals receive the data from the bus controller along the procedure line (LP) after the instruction words.

The exchange system is designed around two digibuses each comprising:

- a unidirectional procedure line (LP), on which the bus controller transmits procedure words to the remote terminals;
- a two-directional data line (LD) on which data are transmitted and received.

Each of the two bus controllers is capable of controlling in turn the two digibuses.

An exchange proceeds in two stages:

Preselection of the peripherals involved in the exchange by means of an instruction message transmitted by the bus controller in operation on the procedure line (LP) of the bus in operation, followed possibly by a delay (response time of the remote terminals).

Transmission and reception of the data: The remote terminal selected by the instruction message to transmit, transmits a data message on the data line (LD) (corresponding to the procedure line used). The remote terminals selected to receive the message then accept these data from the data line.

The minimum period between two exchanges is the duration of one character (10 microseconds).

6.4 Origin and Destination of the Messages (Message routing)

Various types of exchanges can be carried out via the digibus when initiated by a bus controller.

To carry out these exchanges messages are transmitted:

- via the procedure line in the centralised mode from the bus controller to one or more remote terminals (procedure and/or instruction or data);
- via the data line in the centralised mode from one remote terminal to the bus controller (acknowledgment of preselection, data, status†);
- via the data line in the decentralised mode from one remote terminal to one or more of the other remote terminals, including the bus controllers (data and/or procedures).

These exchanges are performed in one of two ways:

- *physical addressing:* Each instruction word transmitted by the bus controller selects explicitly one of the remote terminals involved in the exchange; in this case, the multireception of an item of data implies as many reception instructions as there are receiving remote terminals.
- *label addressing:* In this instance the receiving remote terminals (multireception) are selected implicitly by a single instruction word containing an indication both of the explicit address of the transmitting remote terminal and the name of the information transmitted ("system" address or "label" address).

6.5 Synchronisation of the Remote Terminals

Synchronisation at message level: by envelope detection

Synchronisation at word level: by counting and format checks

Nature of clock: this is extracted from the words moving along the procedure line of the bus in service

Response time of a remote terminal: (See plate 3)

General case: This applies to remote terminals which are always available to carry out a data exchange (send or receive) immediately they receive the instruction word (See plate 3).

Special case: In the opposite case the procedure includes an instruction which prepares the remote terminal or terminals; this instruction is sent well in advance of the actual data exchange command instruction.

Another, more flexible, procedure can be used in cases where the response time is unimportant: delay words can be included after preselection of the remote terminal and before transmission and reception of the data.

* If the "computer" and "bus controller" functions are processed in the same "box", the "computer" is regarded as a remote terminal by the bus controller and in particular it has an address in the data exchange system.

Special functioning in the centralised mode is possible (not used at the present time). In such a case, all the data must pass through the bus controller in service.

† In some cases this message will be a duplicate of one transmitted by the controller in order to enable the latter to check the correct functioning of a coupler.

7. Structure of the Exchanges

7.1 Types of Words or Strings of Characters

The organisation of exchanges via the digibuses requires definitions of the various types of words or strings of characters:

On the data procedure line:

- the preselection instruction words (2 characters);
- the additional instruction word (2 characters);
- the string of delay characters (1 to 256 characters);
- the string of service characters (1 to 256 characters)

On the data line:

- the characters acknowledging receipt of preselection instruction, or echo characters, transmitted by the remote terminals (1 character);
- the string of data or status characters (1 to 256 characters) grouped in words of 1 to 4 characters.

The instruction words are transmitted by the bus controller in operation and at its own initiative. They permit the selection of the remote terminals concerned in the exchange: the transmitting remote terminal and the receiving remote terminal or terminals. These words are also used for initialising the exchange for the remote terminals selected:

- type of addressing and address of data block (send or receive instructions);
- format of data block (additional instruction).

The data words (1 to 4 times 8 useful bits) contain the binary values for the data exchanged between the various remote terminals. These words are always preceded by one or more instruction words (on the procedure line) containing information (by means of the addresses which they include) on the origin and the destination or destinations of the parameters. The data are transmitted in blocks of from 1 to 256 characters.

The status words contain information on the condition of the peripherals which transmit them. They are considered as special words and are generated by the remote terminals on request from the bus controller (specific transmission instruction).

The words acknowledging receipt of the preselection instruction are generated by each remote terminal addressed by an instruction word. They are received solely by the bus controller in operation, which thus checks the remote terminal addressed (correct selection and condition of the remote terminal with respect to the exchanges on the bus).

The string of delay characters generated by the bus controller in operation provides for a certain amount of flexibility in the command in the case of remote terminals with a considerable response time which corresponds to the duration of a few characters (see para. 6.5).

The string of service characters is also generated by the bus controller on the procedure line. It is always preceded by instruction words and, as necessary, by delay characters. The service characters enable the remote terminal selected by the instruction words to act as a transmitter, to decode the clock so that the successive characters in the data block can be transmitted.

7.2 Type of Addressing

Two types of addressing are used in the exchange procedure:

- explicit or physical addressing (direct addressing);
- implicit or "label" addressing (indirect addressing).

The type of addressing used for each exchange depends on the nature of the remote terminal involved and the number of remote terminals which are to receive the data.

Explicit addressing is used when the data to be conveyed involve only a few receiving remote terminals; this type of addressing implies in fact one instruction word per receiving remote terminal concerned.

This type of addressing is nevertheless compatible with an exchange per block words.

In label addressing the remote terminal transmitting the block of data words is addressed in an explicit manner, and the receiving remote terminals in an implicit manner. This type of addressing is very useful for an exchange of data which originate in one remote terminal and are sent to a large number of receiving remote terminals, and also for high speed data exchange.

Label addressing uses only a single preselection instruction word whereas explicit addressing would make it necessary to use one instruction word per receiving remote terminal. To use the label procedure the remote terminals must have stored the labels of the type of data blocks they are likely to receive. If the receiving remote terminal recognises in a label instruction word a block or a data item which concerns it, it accepts this information as soon as it appears in the digibus.

8. STRUCTURE OF THE MESSAGES

8.1 Word Structures

8.1.1 Procedure Line

The messages sent by the control unit along the procedure line are composed of contiguous characters. They are separated by a blank which has a duration of more than one character (10 microseconds). Each procedure message corresponds to an exchange and is generally divided into three sections covering the three phases in the exchange procedure:

- instruction phase for preselection of the equipments involved in the exchange;
- additional instruction and delay phase;
- data phase.

Preselection instruction phase

During this phase which begins at the start of the exchange and finishes before the additional instruction, the bus controller sends out on the procedure line a string of two-character instruction words having the following format: (see Figure 9).

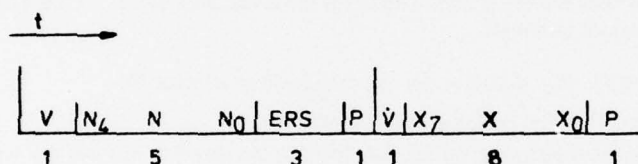


Fig.9 Format of the instruction word

In the fields which have several coded bits, the latter are arranged with the most significant bits at the beginning (as from V)

Each of these instruction words selects a remote terminal which is explicitly designated by the 5 bits in the *N* field (address field) (address of the remote terminal).

The *X* field corresponds to a function or to a data block address or to a channel number.

The coding of the 3-bit *ERS* field or identification field distinguishes 8 different types of instruction:

ERS	Mnemonic code	Meaning
000	CC	Additional instruction (see additional instruction and delay phase)
001	CF	Function instruction coded in the octet X_F for the remote terminal with address N_F
010	CR	Instruction for the remote terminal with address N_R to receive the data block addressed for it by the octet X_R
011	CS	Instruction for the remote terminal with address N_S to receive the service character block addressed for it by the octet X_S
100	CE	Instruction for the remote terminal with address N_E to send the data block addressed for it by the octet X_E (decentralised mode)
101	CV	Channel instruction bit manipulation
110	CL	Label instruction. The octet X_L denotes the label number
111	CT	Instruction to check and test the bus coupler of the remote terminal N_T (slaving of procedure line on the data line)

Function instruction (CF): Instruction for the remote terminal receiving the instruction to prepare a parameter, change the mode, initialise etc. The X_F field indicates the function or functions to be performed.

(The French initials in the mnemonic code have been retained for ease of reference to the Figures).

Instruction to receive (CR): X_R is the address, at the level of the receiving remote terminal N_R , of the data or data block to be received.

Service instruction (CS): X_S is the address, at the level of the receiving remote terminal N_S , of the string of service characters which it receives on the procedure line from the bus controller in operation (UG_1 or UG_2).

Instruction to transmit (CE): X_E is the address, at the level of the transmitting remote terminal N_E , of the data or data block to be sent.

Channel instruction (CV): X_V indicates for the remote terminal of address N_V :

- 8 bits received on the procedure line in the character C of the additional instruction (see additional instruction and delay phase);
- 8 discrete signals to be sent out on the data line.

Label instruction: This is an explicit instruction to transmit, for the remote terminal of address N_E and an implicit instruction to receive. The 8 bits in the label (X_L field) indicate the data name or the block data name, this name being a "system" address; the receiving remote terminals involved in the data item or data block which follows this instruction word recognise it by decoding the 8 bits in the label (there is therefore an *implicit instruction to receive* for the remote terminals concerned).

Note: In the case of a label instruction, this may simply be an instruction intended for several remote terminals who recognise implicitly that they are the receiving units for the instruction word: in this case the N field is not used (number of a non-existent remote terminal).

Check test instruction (CT): X_T indicates, for the peripheral of address N_T :

- a string of service characters received on the procedure line;
- a string of test characters transmitted on the data line (in the majority of cases this will be identical to the string received on the procedure line).

The V bit (bit which validates the request for transmission by the transmitting equipment) is always in the "O" state during this phase.

Each P bit indicates the odd parity corresponding to the character of which it forms part. The bits are processed by the bus controller in service and when received represent the satisfactory condition of the transmission (no interference).

The string of instruction words is made up of any assembly of the six types of words: CE, CR, CS, CV, CL or CF. The only restriction is that there should not be several instructions (CE, CV or CL) denoting different sending terminals in the same exchange.

Additional instruction and delay phase

This phase, which regularly begins with the additional instruction word, may, as an option, contain a string of delay characters generated by the bus controller.

The additional instruction word generated by the control words consists of 2 characters and has the following format: (see Figure 10).

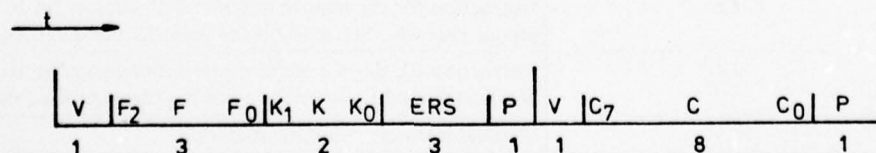


Fig.10 Format of the additional instruction word

The decoding of the ERS = 000 field stops the preselection instruction phase. This word serves as an addition to all

the remote terminals previously addressed in the preselection instruction phase. The instruction is specified by the F, K and C fields.

For the remote terminals instructed to send or to receive:

- C indicates the number of characters in the data block or in the string of service characters ($C + 1$ characters);
- K indicates the format of the data words ($K + 1$ characters);
- F contains 3 function bits to be specified.

For the peripherals addressed by a channel instruction:

- C is an immediate factor of 8 bits for setting or resetting (at "1" or "0") 8 separate signals (all together or one by one);
- K indicates the effect of C according to the following coding:

K	Effect
00	non operation (NOP)
01	reset following mask C
10	set following mask C
11	set or reset according to C

The string of delay characters is generated by the bus controller, as necessary, in order to delay the start of the data phase; The delay characters have the following format:

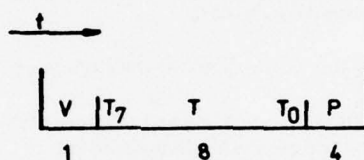


Fig.11 Format of the delay character

The content of these characters is transmitted to the remote terminal but the latter must not use it, except in special cases.

The V bits in the additional instruction word and those in the delay characters have the same meaning as in the preselection instruction phase. They set at the penultimate character in the additional instruction and delay phase.

Data phase

If this phase exists the bus controller transmits on the procedure line a string of $C + 2$ service characters. Each service character has the following format: (see Figure 12).

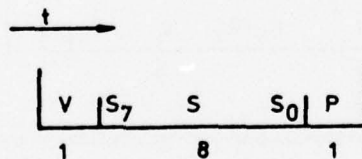


Fig.12 Format of the service character

During this phase the V bit is set except for the last three service characters, thus making it possible to stop the transmission of the data at the level of the transmitting unit. This bit is generated by the bus-coupler. The content of the service characters is used by the remote terminals which have been addressed by a service instruction. These characters are also used by the bus-coupler of the transmitting unit to trigger bit by bit the transmission of the data characters on the relevant data line.

8.1.2 Data line

The messages circulating on the data line are composed of preselection acknowledgment characters from the equipments selected, and data or condition characters.

Note: The number of characters in the data block is $C + 1$ (1 to 256 characters with $C = 0$ to 255). The procedure includes an extra service character to facilitate the COS-CSS dialogue (presence of a clock pulse after deserialisation of the last useful character received).

The data words comprise $K + 1$ characters (1 to 4) having the following format: (see Figure 13).

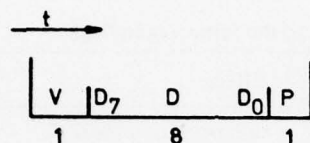


Fig.13 Format of the data word

They are initiated bit by bit from the service characters during the data phase.

The P bits have the same meaning as previously (odd parity of the character of which they form part). They are generated by the bus-coupler of the unit transmitting the data.

The 8 bits in the D fields indicate the value of the data (*most significant bits at the beginning*).

The V bit ($V = 1$) validates the data. In the event of a non-valid data element the receiving equipment does not take this into account. This bit represents the correct functioning condition of the circuits which generated the data and of the input/output circuits of the equipment. It is generated by the transmitting equipment proper.

The format of the words and the odd parity of each data character are checked by the bus-coupler; in the event of an error being detected a message cancellation signal is sent to the CSS (subsystem coupler) to invalidate the use of the data by the remote terminal.

The status characters transmitted on the data line during the data phase have the same structure as the data words and are considered by the exchange system as containing a particular data item. They are always addressed explicitly by means of a specific exchange.

The preselection acknowledgment characters (*echos*) have the following format: (see Figure 14).

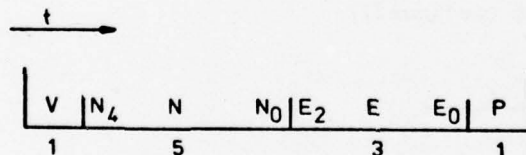


Fig.14 Format of the preselection acknowledgment characters

The bit $V = 1$ indicates the validity of the digibus, as seen by the bus-coupler of the remote terminal N .

The 5 bits in the N field contain the address of the remote terminal transmitting the word. The field must be identical with that of the instruction from which it originated. The remote terminal having the address N places in the 3-bit E field 2 bits showing its status regarding the transmission system (interrupt request, summary of failures); the 3rd bit is a reserve bit.

The P bit indicates the odd parity of the preceding octet. The preselection acknowledgment character is generated when the bus coupler recognises the number of the peripheral (N field).

8.2 Performance of Exchanges

Plate 1 contains some examples of exchanges. *Example Figure 1* shows the reception of an instruction by the remote terminal with the address N_F .

Example Figure 2 shows the exchange of a data item or of a data block between the transmitting remote terminal (address N_E) and one or more receiving peripherals (address $N_{R1} \dots N_{Ri}$).

In the case of the exchange of a data element or of a data block, the X fields of the transmission or reception instruction words correspond to the explicit address, for the remote terminal selected (sub-addressing of the data element or data block to be transmitted or received).

Example Figure 3 corresponds to the exchange of a block of data which are implicitly addressed for each remote terminal by a label (system address X_L).

Note: It is important to note in the frame that the respective positions of the various data blocks and data elements is always the same. Certain additional words can, however, be inserted in the basic frame: instruction words, status words etc.

In addition, the data making up each data block have a well defined position in the block concerned; the latter is never changed from one frame to another.

In any one exchange the data words have the same number of characters ($K + 1 = 1$ to 4).

8.3 Overall (Synoptic) View of the Multiplex Exchange

An overall picture of the exchange system with double redundancy is shown in Plate 2.

9. MONITORING OF THE EXCHANGE SYSTEM (PROVISIONAL)

Several types of monitoring are carried out at various levels:

- monitoring of the exchanges at the remote terminal level;
- monitoring of the exchanges at main bus controller level;
- monitoring of the main bus controller by the stand-by bus controller.

9.1 Monitoring of Each Remote Terminal

On recognising its address in an instruction word, the remote terminal in conjunction with its bus-coupler, carries out a number of checks on the two lines of the particular digibus on which it is being interrogated (transmission or reception):

Procedure line

- checks of the format of the characters received (bit rate, number of bits)
- odd parity check of the characters

Data line (reception)

- checks of the format of the data characters
- bit rate, number of bits
- odd parity check of the characters

If an error is detected during one of these checks the remote terminal must regard the exchange as invalid. It must then indicate in its status word the fault found in regard to the exchange, for subsequent transmission to the bus controller in service.

9.2 Monitoring of the Main Bus Controller

The function of the main bus controller is to control the exchange of data by transmitting the various instruction words; service and delay characters on the procedure line of the digibus in service.

It has, however, to check that the exchanges on the bus in service are proceeding satisfactorily so as to detect any malfunctioning (of the digibus or of a peripheral).

To do this, the bus controller checks by re-looping:*

- the format of the words which it transmits on the procedure line of the bus bar

* Re-looping: the bus controller receives from the digibus the words it has sent.

- their bit rate and the number of bits
- their odd parity bit.

The main bus controller receives the preselection acknowledgments from the remote terminals selected by each exchange and has to check:

- the address N of the preselection acknowledgments
- the preselection acknowledgment field E
- the validity bits of the bus and the odd parity bits.

Other checks can be made by the computer which may be associated with the bus controller, for example:

- a check of the addressing of certain parameters per programme
- a probability check and/or validity check of certain parameters.

The bus controller has two other major functions; these relate to the safety and the reliability of the exchange system and are as follows:

- failure test: very sophisticated failure detection circuits enable the main bus controller to carry out automatic tests up to and including the level of the bus-coupler. The discrete "failure" signal is fed direct (independently of the bus) to the standby bus controller.
- standby digibus test: at a relatively low frequency the main bus controller has to switch the exchanges over the standby bus in order to test that it is functioning correctly. To do this it feeds instruction words to the latter for the purpose of interrogating all the peripherals (or some of them); if an incorrect reply is received, the main bus controller invalidates the standby bus.

Other direct wires between the two bus controllers enable the standby control unit to be given information about the number of the bus on which for example the main bus controller is operating; various service signals will have to be specified later on (discrete mode change signals, special requests from the remote terminals).

9.3 Monitoring of the Main Bus Controller by the Standby Bus Controller

The standby bus controller monitors the main bus controller in the following two ways:

- telemonitoring: direct connection between the two bus controllers;
- a more general test by monitoring the exchanges on the bus in service or by carrying out cyclic check exchanges itself (system test sequence in particular).

Telemonitoring

The discrete failure signal from the main bus controller enables the standby bus controller to inhibit the defective controller and to continue the data exchanges.

General test

Monitoring the bus

The standby bus controller can monitor the 2 bus-bars. The tests which it makes are similar to those specified for the main bus controller, in Paragraph 9.2.

It could also check that for certain exchanges the block addressed is in fact identical for all the remote terminals concerned. For example, the standby bus controller could check certain sequences of instruction words generated by the main bus controller, by comparison.

Cyclic control by the standby bus controller

The standby bus controller can check that certain data exchanges which are identical in the main mode and in the standby mode are proceeding correctly by controlling them itself in a cyclic manner. In this case it should be possible for the two bus controllers to be synchronised with each other.

10. RECONFIGURATION

10.1 Failure of a Remote Terminal

Whenever a case of malfunctioning is discovered during checks by the main bus controller, the latter can, in conjunction with a computer (division of tasks to be specified later), command the repetition of certain messages, the request for the transmission of status words to any remote terminals about whose functioning there is some doubt or the change of mode requested by a remote terminal in the E field of the acknowledgment words.

Following this type of exchange and after processing has taken place, the bus controller can send instructions to other remote terminals (input inhibitions, control of failures, display of data, storage of status words and of certain flight parameters.).

In the absence of a reply from a remote terminal on the bus (failure of remote terminal, line cut . . .) and after the above checks, the bus controller switches over to the other bus and continues the exchange of information.

10.2 Failure of the Digibus

Any failure of the bus in service is detected by the bus controller which then switches over to the other bus. The faulty bus is deactivated by the absence of a message on its procedure line. If, in spite of this, an uncontrolled continuous transmission occurs in one digibus or both, the bus controller deactivates the transmitting remote terminal by means of a function instruction word (on the procedure line), which invalidates its interface circuits.

The pilot's data display shows:

- the bus in service
- any failures of the buses.

Furthermore, action by the pilot can change the bus in service (unless a failure has been detected in one of the two buses).

The initial bus in service will be initialized before each flight by the computer after the two buses have been checked by the control units. (bus controllers)

10.3 Failure of a Bus Controller

Assume that UG 1 is the main bus controller and that UG2 is the standby control unit. Several types of failure can be considered.

	ACTION
Failure of UG1 : detected by UG1	<ul style="list-style-type: none"> — UG2 takes over control; — display of the failure to the pilot.
Failure of UG1 : not detected by UG1	<ul style="list-style-type: none"> — display to the pilot; — warning certain equipments; — pilot's decision, as necessary
Failure of UG2 : detected by UG2	<ul style="list-style-type: none"> — UG1 takes over control; — display to the pilot.
Failure of UG2 : detected by UG1	<ul style="list-style-type: none"> — UG1 takes over control; — display to the pilot.

The bus controller in service is displayed to the pilot. Also, in certain cases the pilot is able to bring UG2 into operation.

11. MAINTENANCE PREPARATION

The existence of a multiplex digital data transmission system in an aircraft is to make it possible, without any other external means, to locate the particular "black box" which has failed.

The control system and the structure of the exchanges have been designed to facilitate maintenance.

The use of the "function" instruction words and the preselection acknowledgments enables the status of the equipments to be recorded in flight.

The running of certain test programs in flight and, as necessary, in a more comprehensive manner on the ground permits the application of the status results recorded.

The status words to test the condition of the various remote terminals in the system are requested in several ways depending on the type of remote terminal concerned; for example:

- cyclically in the case of remote terminals whose function is solely to receive data;
- at a very low rate when testing certain transmitting and receiving equipments;
- when the control program is interrupted following the detection of a fault (acknowledgments etc.).

Similarly, certain flight parameters on the digibus are systematically recorded in flight (attitude, aerodynamic parameters...) over a certain period so that the flight phase during which the fault occurred can be reconstituted on the ground.

12. DEFINITION OF THE SUB-SYSTEM JUNCTION OF THE BUS-COUPLER

12.1 Definition of the Bus-Coupler

The bus-coupler connects each remote terminal to the digibus, as described in Section 1.

The bus-coupler provides the standard functions which have to be performed by all the remote terminals connected to the redundant digibus. The special or specific-to-type functions will be performed by the sub-system couplers.

The standard bus-coupler, as defined in this Section, is referred to as the COS (Coupleur de bus Standard).

It performs the following functions in particular:

- reception, demodulation and checking of the formats of the procedure messages sent out by the bus controller along the procedure lines;
- separation, recognition, deserialisation and odd parity checks of the procedure characters (procedure lines);
- recognition of the type of exchange (function instruction words, transmission, reception);
- recognition of the type of addressing (physical or label addressing);
- transmission along the data line of an acknowledgment word (when the actual remote terminal unit decodes the address);
- reception, demodulation and checks of the format of the data messages on the data lines;
- separation, deserialisation and odd parity checks of the characters in the data words received;
- transfer of these characters one by one to the sub-system coupler;
- calculation of the odd parity of the data octets supplied by the sub-system coupler in the case of a transmission;
- serialisation and generation of the data message;
- modulation and transmission of the data message.

12.2 Definition of the Standard Interface

The COS (Standard bus coupler) decodes the exchange procedure in the bus, as described in the preceding paragraphs: it performs the functions specified above and makes the connection to the bus of the sub-system coupler by means of a standard interface (sub-system coupling).

The interface signals are separated by function into 3 groups (not including the power supply signals):

- information signals (2 unidirectional octet bus mainly);
- time base and dialogue signals;
- status signals (error signals, functioning modes).

Plate 4 shows the interface signals with the sub-system coupler.

12.3 Dialogue Procedure

The procedure words (received on the procedure line) and the data words (received on the data line) are demodulated and deserialised by the COS (standard bus coupler). Only the octets of these words which are useful to the sub-system coupler are transmitted to it along a receive octet bus (B_R) controlled by the control logic means of the relevant dialogue signals.

All the words transmitted by the COS (standard bus coupler) (acknowledgment words and data words) on the data line of the digibus are supplied by the sub-system coupler along a send octet bus (B_E) controlled by the control logic (relevant dialogue signals). No transmission can be made on the bus (transmission of acknowledgments or data) if the control signal AE (authorisation to transmit on the data line) is in the "1" logic state.

The signal VB (validity of the bus) fed to the sub-system coupler by the COS switches to the "0" state when the latter detects a transmission error on the bus lines (message format, parity error).

- A. *General information and procedure for the exchange of an octet between the bus-coupler and the sub-system coupler (Plate 5).*

A.1 Reception of an Octet by the Sub-System Coupler

The receive octet bus (BR) is associated with a number of signals transmitted by the COS to the sub-system coupler:

- a dialogue signal
VR: reception validation
 - a data signal
VMR: (validity of the word received): a signal associated with the receive bus (BR) which indicates the validity of the word of which the data octet received forms part. It must be taken during the data phase by the receiving remote terminal only (VDR = 1) .
Duration of the VR signal: approximately 5 micro-seconds.
1. The position of the bits in the various characters in the procedure on the receive octet bus BR and on the send octet bus BE is shown on Plate 9.

A.2 Transmission of an Octet by the Sub-System Coupler

The "send octet" bus (BE) is associated with:

two dialogue signals

DE: requesting transmission: COS CSS
VE: validation of transmission: CSS COS

one data signal

VME: validity of the word transmitted: CSS COS

Using the signal DE, the COS sends the sub-system coupler a request for an octet to be transmitted. The sub-system coupler then generates this octet on the parallel send bus (BE), as well as the validity of the relevant word by means of VME. It then replies to the DE signal by sending the signal VE validating the octet, within a maximum period of 3 microseconds. The signal VE must therefore have a duration of at least 1 microsecond.

The VE signal must not exceed by more than 1 microsecond the DE signal (risk of overlapping with a subsequent DE signal).

Duration of the DE signal: approximately 5 microseconds.

A.3 Dialogue Clock Pulses (HD – Horloge de dialogue)

The HD signal supplies the CSS with two pulses of approximately 1 microsecond each during the duration of each VR or DE signal (even when they do not exist). The CSS can use one or both pulses when replying (RL or VE).

B. Procedure for Connection to the Bus (Plate 6)

The DC signal (demande de connexion: request for connection) rises approximately 10 microseconds before the beginning of the preselection instruction phase and falls when the exchange has been completed for the peripheral (at the same time as VC (validation commande: validation of instruction) for a non address peripheral).

B.1 Preselection Instruction Phase

The sub-system coupler is connected to the bus as necessary during the instruction phase of the exchange; this phase is identified by the signal VC = 1 (validation of the instruction phase) fed by the COS to the sub-system coupler.

On receiving each instruction word on the procedure line, the COS establishes the dialogue with the sub-system coupler in order to determine whether the peripheral is involved in this word. It should be noted that a peripheral can be addressed more than once during the same exchange; only the last addressing should be taken into account in the continuation of the exchange.*

The dialogue between the COS and the sub-system coupler can be established in two ways, depending on the addressing used in the instruction word:

- explicit addressing:
function instruction word
send instruction word

* A function instruction (CF – commande fonction) does not cancel the effect of a previous instruction.

receive instruction word
label instruction word (for the transmitting peripheral)

- implicit addressing:
label instruction word (for the receiving peripherals)

B.1.1 Explicit Addressing of the Peripheral (Plate 7)

Initially ($HC = 1$) the COS sends the sub-system coupler the first character of the instruction word received and the latter transmits to the COS the preselection acknowledgment character.

The procedure used in both directions is in accordance with that described in Paragraph A.

On receiving the preselection acknowledgment character the COS compares the number N of the peripheral received in the instruction word with that supplied by the sub-system coupler in the preselection acknowledgment character.

If there is coincidence the COS then transmits ($HC = 0$) the second character of the instruction word to the sub-system coupler and stores the acknowledgment octet for transmission (see bus-bar procedure).

B.1.2 The Peripheral Which Receives an Item of Data Addressed by Label (Plate 8)

In this case, in the preliminary stage of the dialogue ($HC = 1$) between the COS and the sub-system coupler, there has been no recognition of the number of the peripheral (see C below).

During the second stage ($HC = 0$) the COS nevertheless transmits the label (second character in the label instruction word) to the sub-system coupler.

The latter replies, if the label number involves it (unit receiving data implicitly addressed by their label), by setting at "0" the signal RL (label reception) for at least 1 microsecond during the length of the VR signal. The COS then stores the fact there will be subsequent reception of the data block.

B.2 Additional Instruction and Delay Phase

B.2.1 Additional Instruction Word

This initializes the functioning of the peripheral as a transmitter or a receiver.

It is characterized by the ERS code: 000.

This word is transmitted in two stages to the sub-system coupler under the procedure defined in A.1, if there has been recognition of an address for at least one of the preceding instructions (see B.1.1).

During transmission of the first character ($HC = 1$) the COS sends a 5-bit instruction (F, K) and $ERS = 000$.

During transmission of the second character the bus-coupler supplies the number of data characters C making up the data block exchanged.

B.2.2 Delay Word

As stated in the procedure for exchanges on the digibus, the delay characters are transmitted to the sub-system coupler with the signal $HC = 0$; *a priori* they are not used by the sub-system coupler.

B.3 Data Phase

This phase is characterised by the signal $VDE = 1$ (validity of transmission request) for the transmitting peripheral and $VDR = 1$ (validity of reception request) for the receiving peripheral or peripherals.

B.3.1 Transmitting Peripherals ($VDE = 1$)

In reply to each request from the standard coupler (COS) the sub-system coupler must supply the successive octets in the data words to be transmitted, and the relevant VME (validity of the word transmitted).

B.3.2 Receiving Peripheral ($VDR = 1$)

The COS supplies the successive octets of the data words received and their validity (VMR) (validity of the words received). The HC signal is in the zero state during this time.

C. *Special Functioning*

The permanent setting at 0 of the RL (label reception) signal forces the COS to receive. This forcing does not produce any additional echo on the data line and its sole effect is to compel the COS to send the CSS all the characters it receives.

12.4 Electrical Characteristics COS/CSS Interface Signals

All the data, dialogue, procedure and monitoring signals are TTL compatible logic signals:

Low level 0 \leq V_L \leq 0.4 volts
 High level 2.4 \leq V_H \leq 5 volts.

The COS (standard bus coupler) output signals are capable of controlling 8 normal TTL loads:

- current consumed by the COS in the "0" state 12.8 mA.
- current supplied by the COS in the "1" state 320 micro-amps.

All the COS (standard bus coupler) input signals, except the two AE signals (authorisation to transmit on the data line) must be able to control 1 normal TTL load:

- current supplied by the COS in the "0" state 1.6 mA
- current consumed by the COS in the "1" state 40 micro-amps.

The two AE signals must be capable of controlling two normal TTL loads:

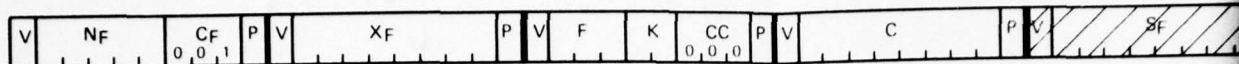
- current supplied by the COS in the "0" state 3.2 mA.
- current consumed by the COS in the "1" state 80 micro-amps.

Power supplies

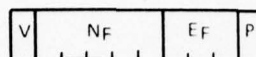
The power consumption for the various power supply units required for the COS (standard bus coupler) is:

- 5 V supply 4W
- + 15V supply 3W
- - 15V supply 3W

The + 15V and the - 15V voltages require $\pm 2\%$ regulation, the + 5V voltage requires $\pm 5\%$ regulation.

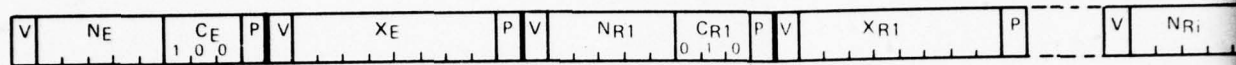


bit string on the procedure line

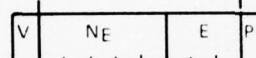


bit string on the data line

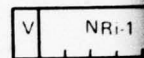
A) Exchange of a function instruction



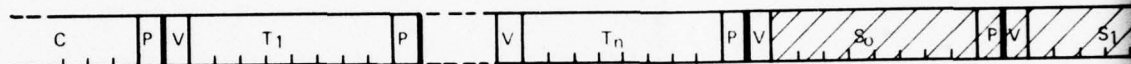
bit string on the procedure line



bit string on the data line

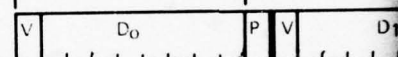


continued



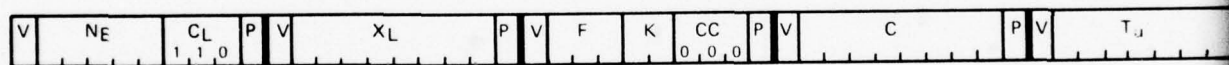
bit string on the procedure line

continued

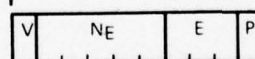


bit string on the data line

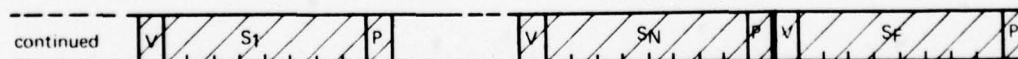
B) Exchange of a data block by explicit addressing



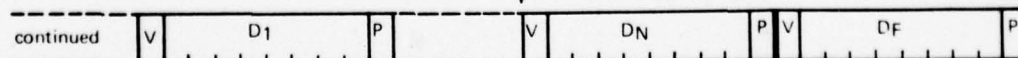
bit string on the procedure line



bit string on the data line

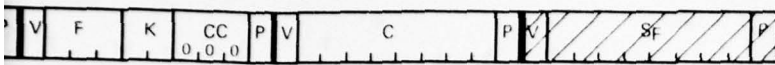


bit string on the procedure line



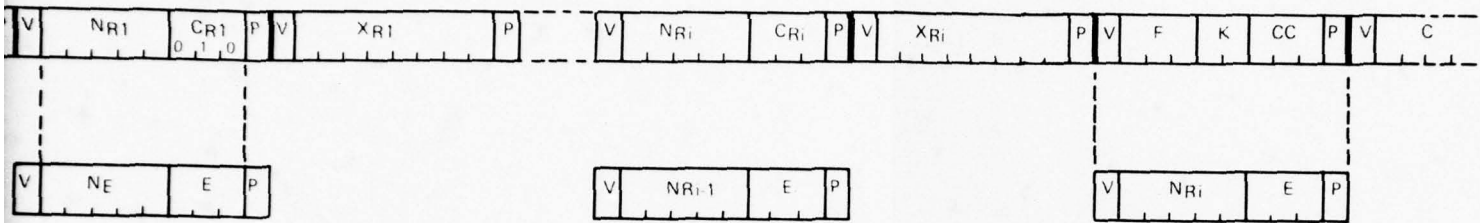
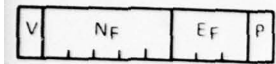
bit string on the data line

C) Exchange of a data block by implicit addressing (label)

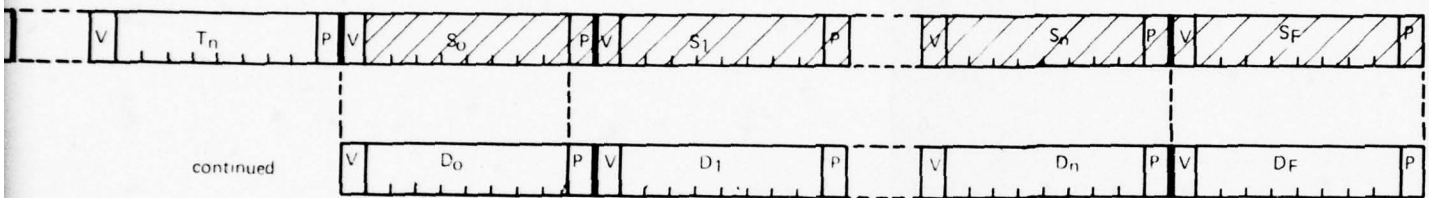


characters not seen by the subsystem coupler

- C_F Function instruction word
- C_E Transmission instruction word
- C_R Reception instruction word
- T Delay character
- S Service character
- C_L Label instruction word
- D Data character
- S_F, D_F End of service / data character



bit string on the data line



bit string on the data line

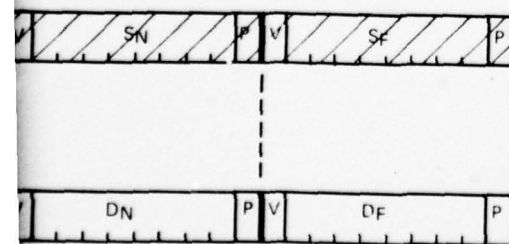
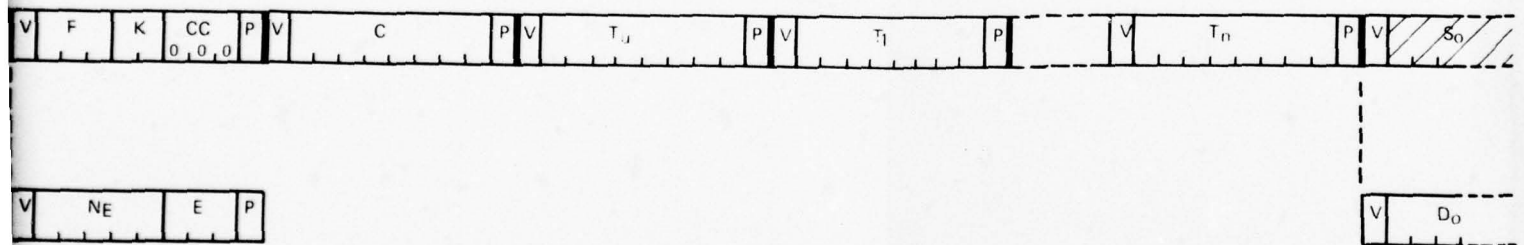
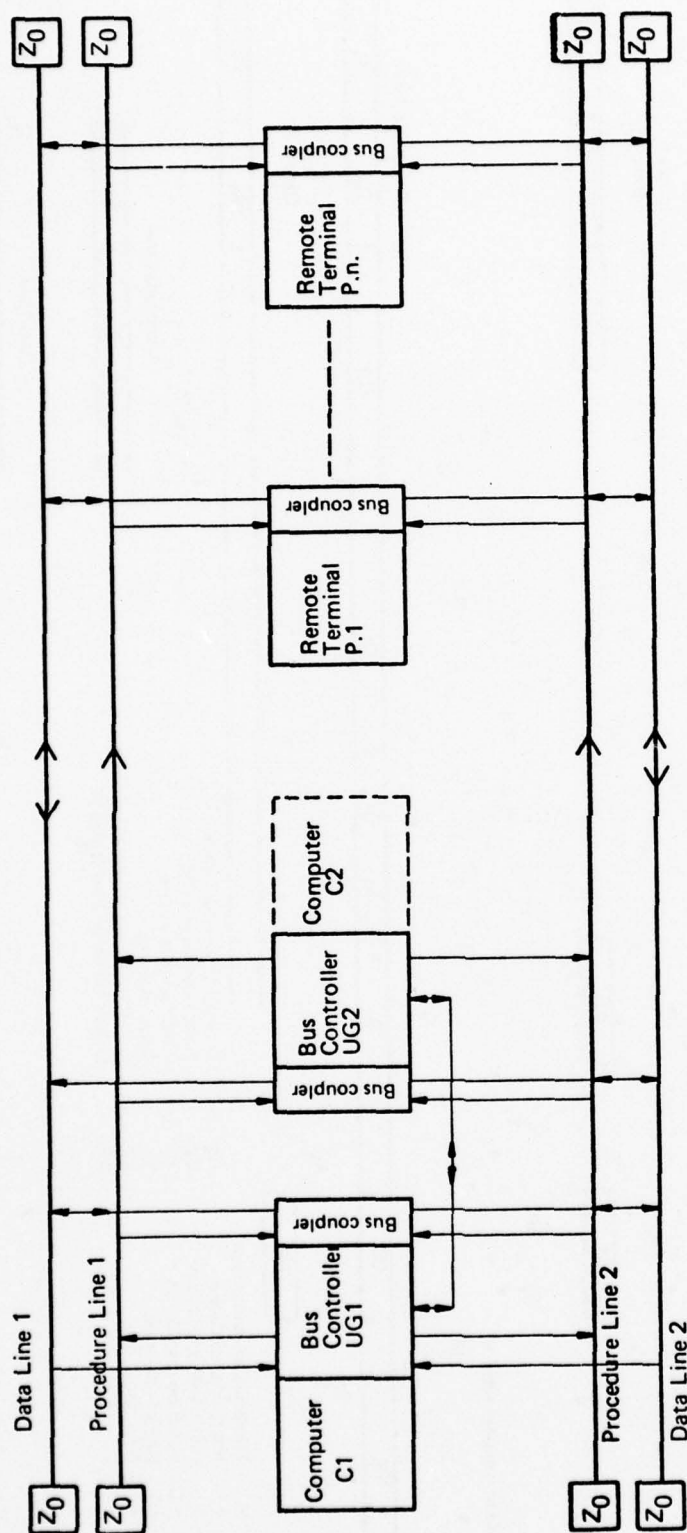
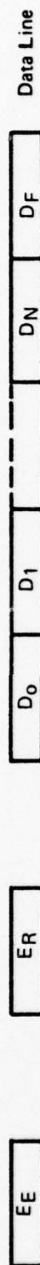


Plate 1 Procedures for the exchange via the digibus

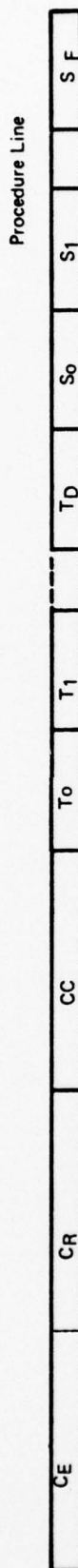


Characteristic Impedance

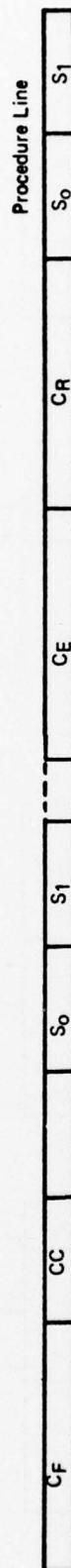
Plate 2 Organisation of the exchange system



a) short response time



Mean response time



CE = Transmission instruction
 CR = Reception instruction
 CC = Additional instruction
 CF = Function instruction
 S = Service character
 D = Data Character

EE = Transmitting status word
 ER = Reception status word
 EF = Function status word
 T = Delay character
 SF = End of exchange service character
 DF = End of exchange data character

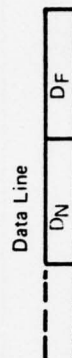
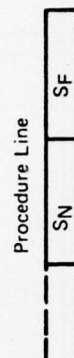


Plate 3 Procedures used for various response times of the remote terminals

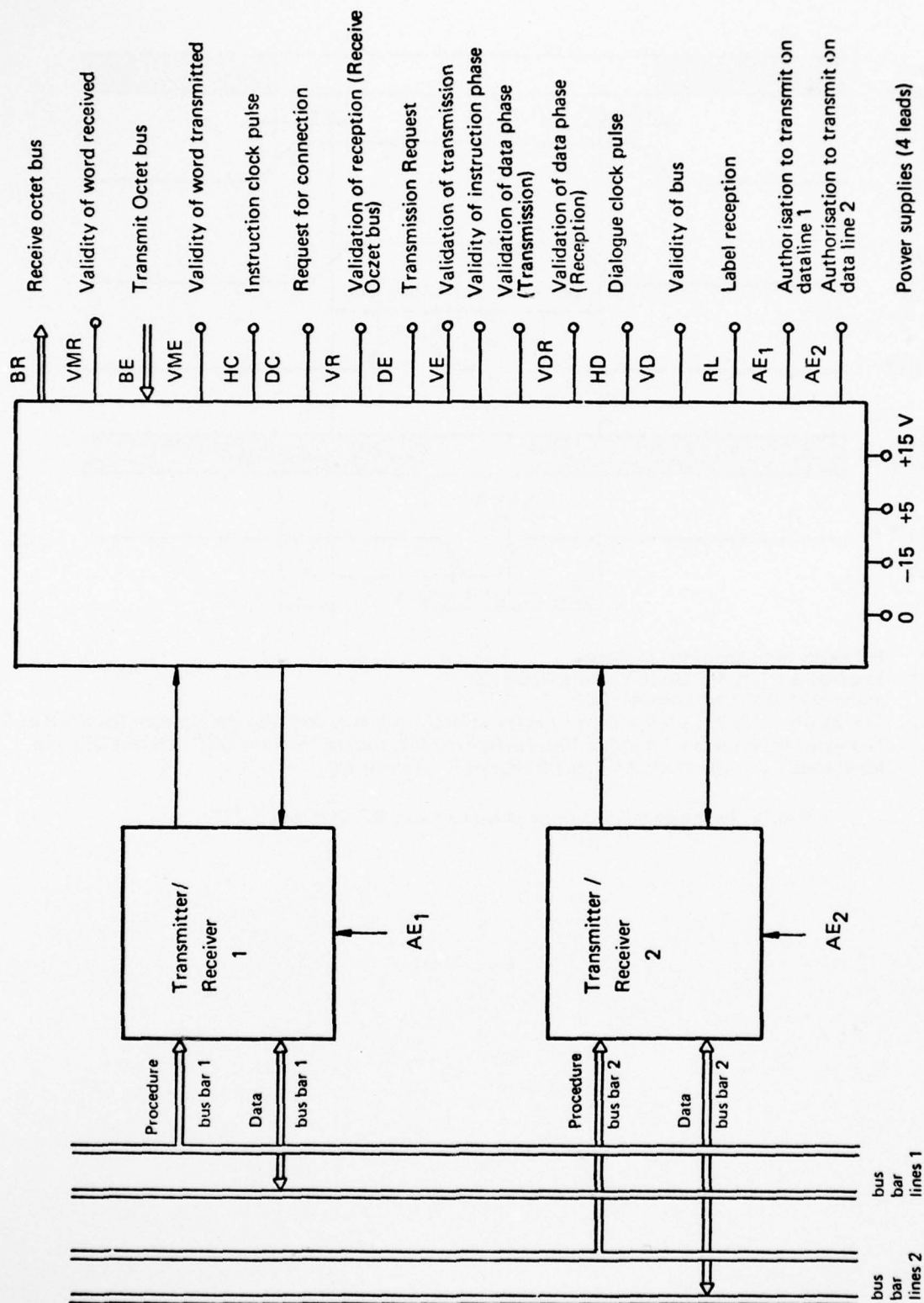
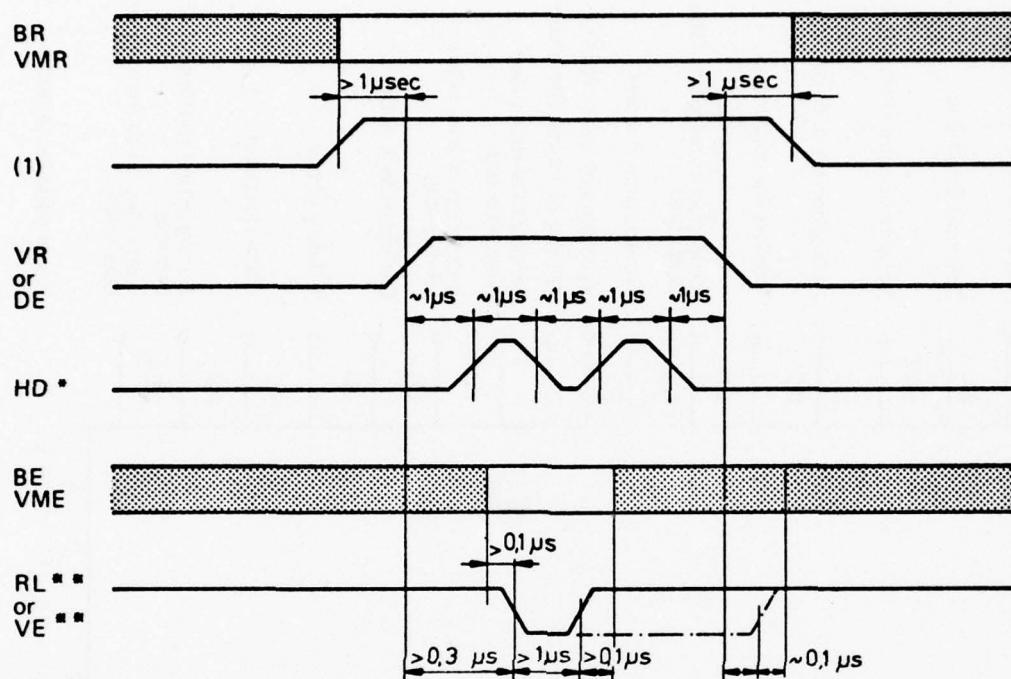


Plate 4 Standard bus coupler block diagram and connections

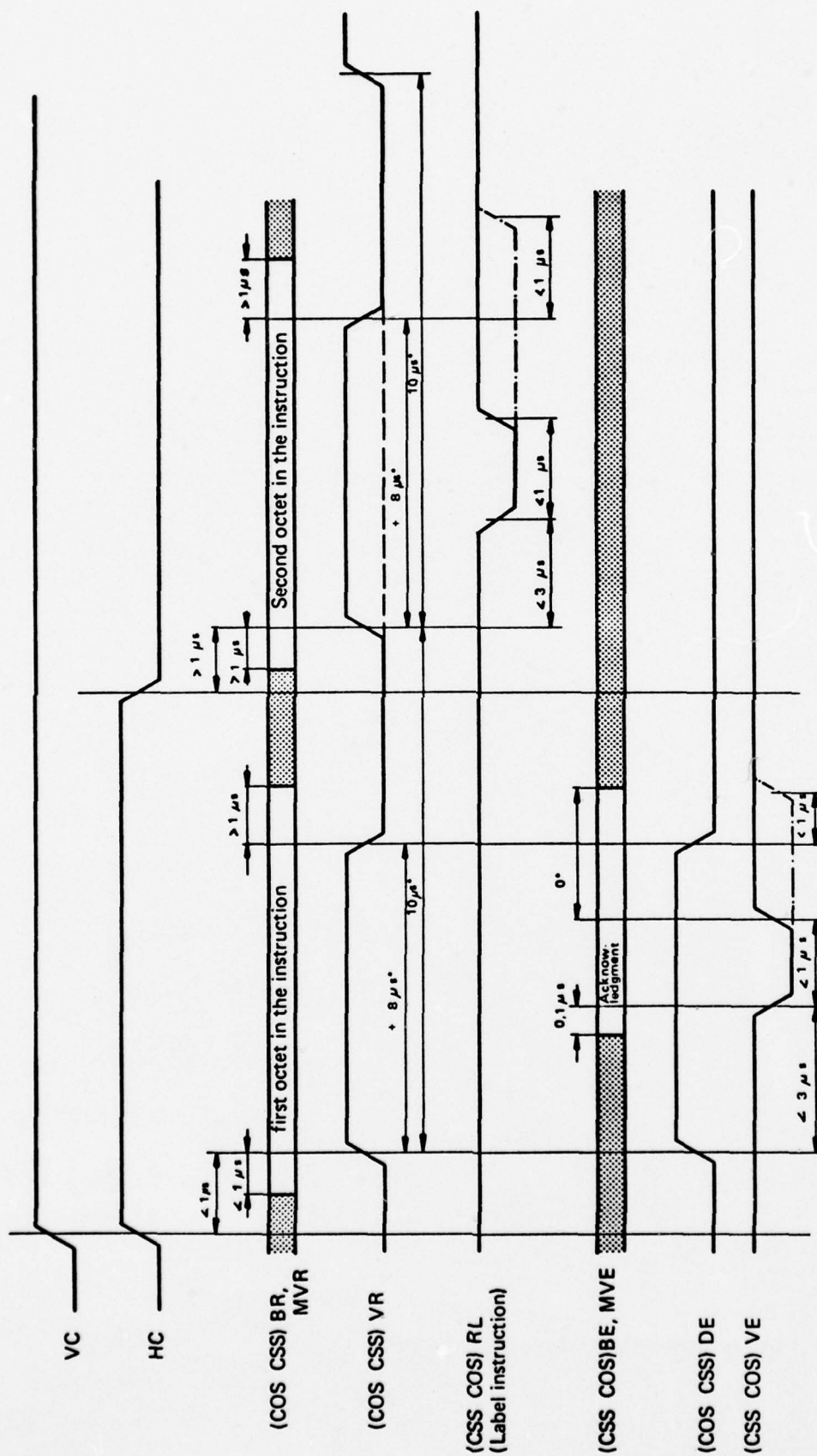


* HD exists throughout the exchange

** The replies RL or VE can be obtained from HD at the level of the bus coupler (COS)

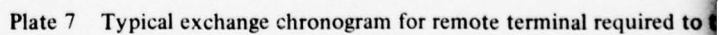
- (1) The signals VC, VDE, VDR, HC are valid for at least $1\mu\text{s}$ before and after the dialogue signal VR or DE. The signal DC is present for about $10\mu\text{s}$ before the first request from the COS (VR and DE) and for at least $1\mu\text{s}$ after the last VR or DE request in an exchange.

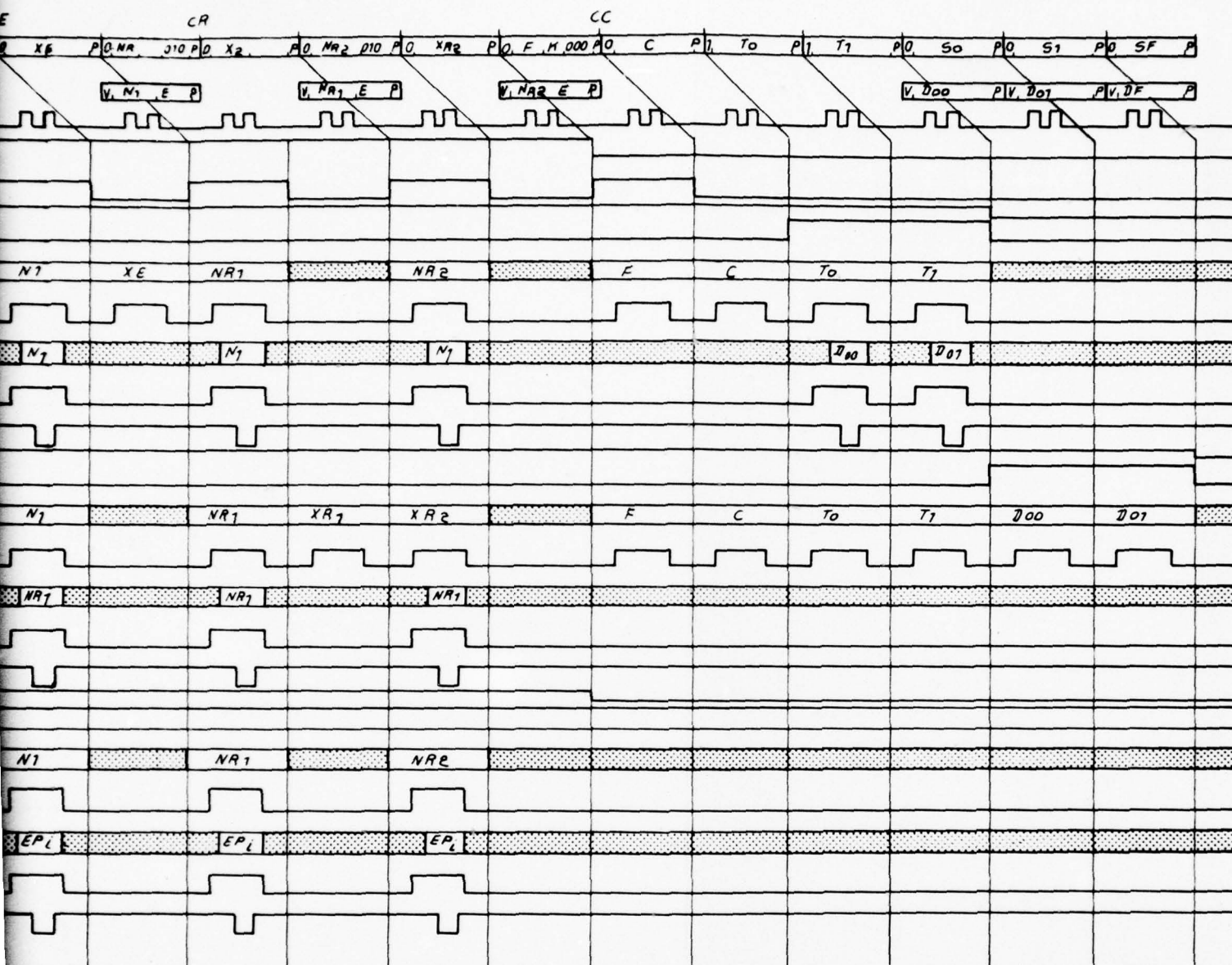
Plate 5 Procedure for character exchange between the COS and the CSS



--- When there is no recognition of an address, except in the case of a label instruction
 * COS input and output

Plate 6 Procedure for the connection of the CSS to the bus instruction phase





change chronogram for remote terminal required to transmit, receive and not addressed

(1)

(2)

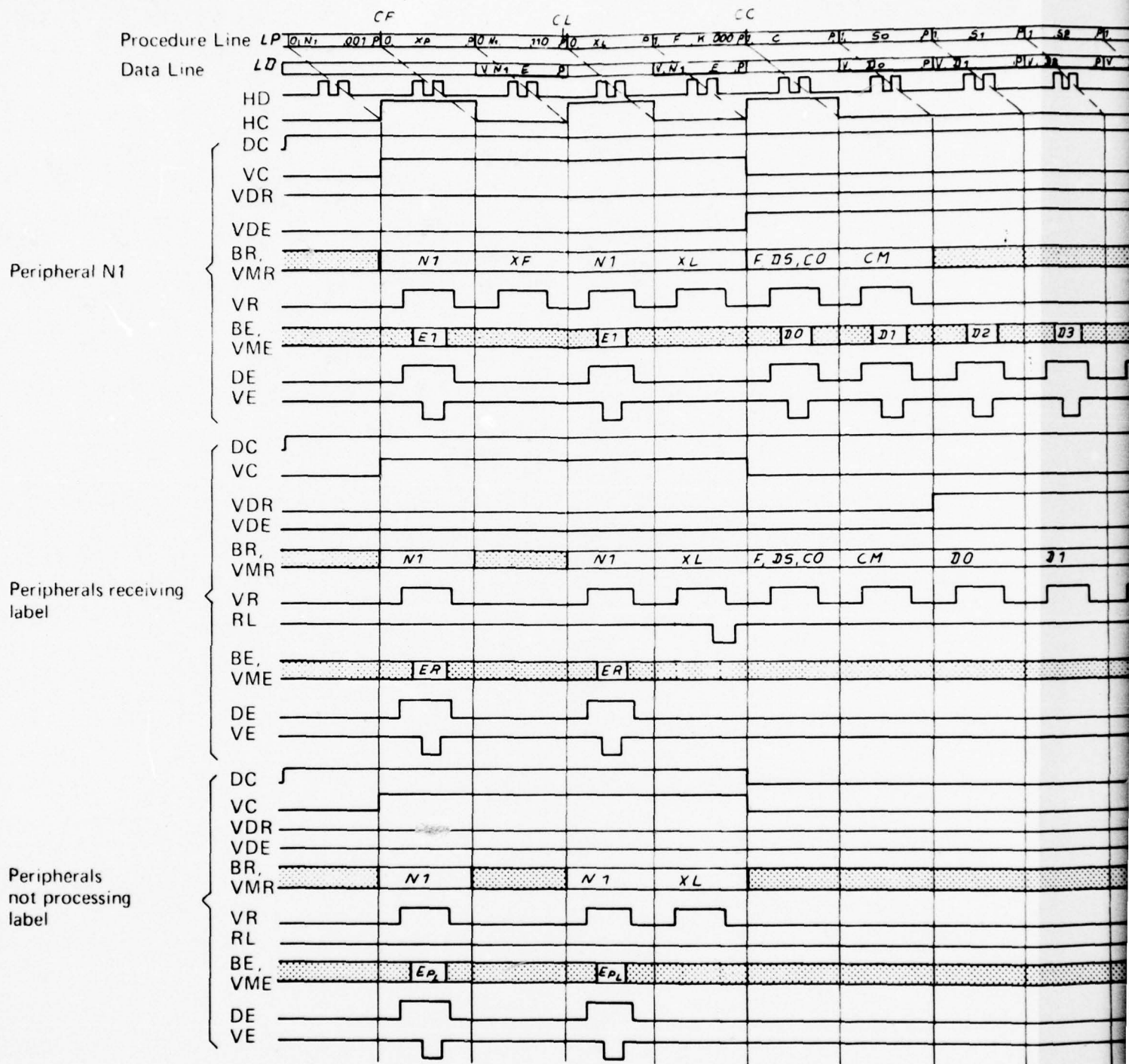


Plate 8 Typical exchange chronogram for remote terminals to transmit and receive by label addressing

47

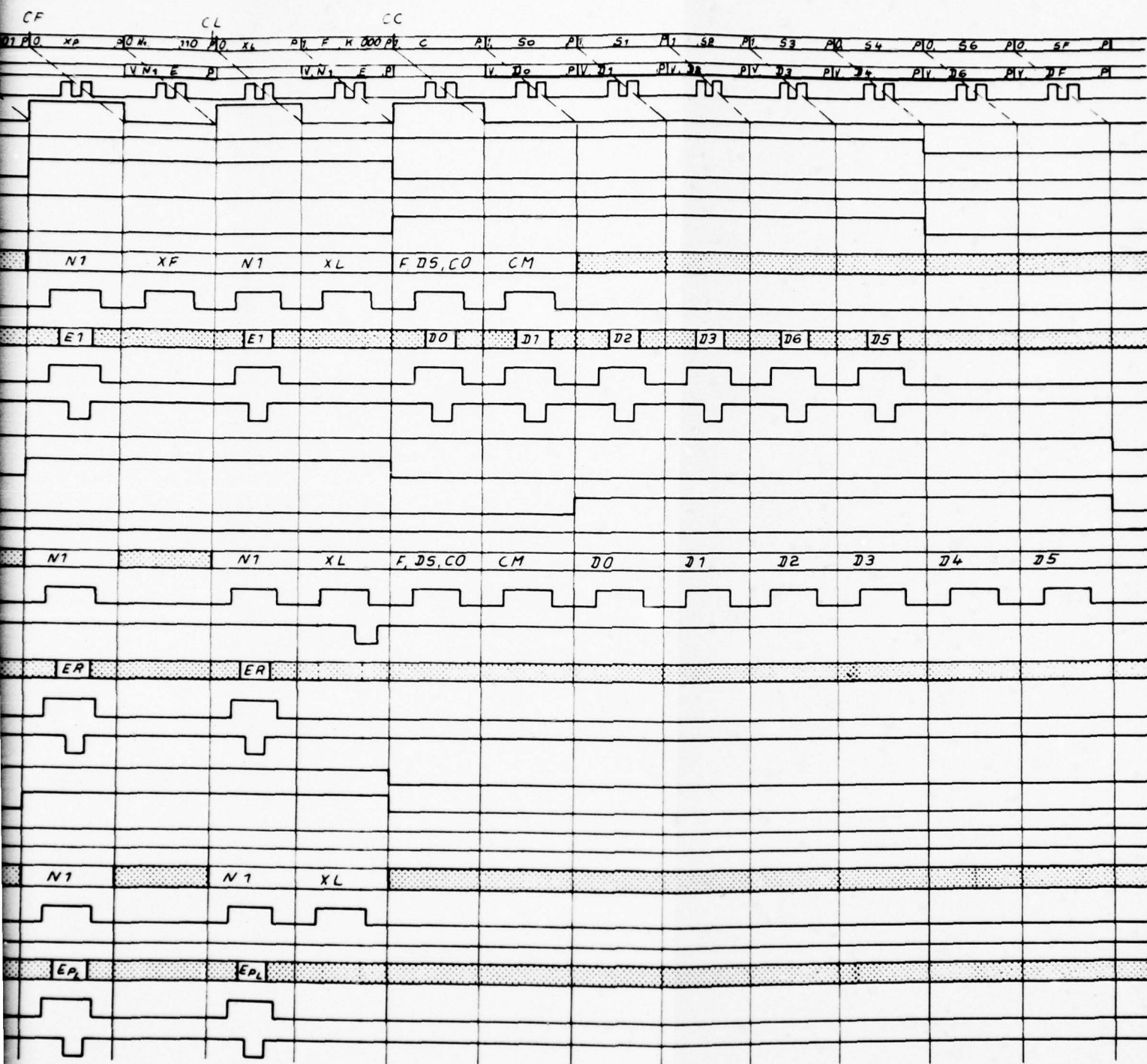


Plate 8 Typical exchange chronogram for remote terminals to transmit and receive by label addressing

4

21

Receive bus

V _{MR}	BR ₇	BR ₆	BR ₅	BR ₄	BR ₃	BR ₂	BR ₁	BR ₀
-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

1st character of instruction words (HC = 1)

V	N ₄	N ₃	N ₂	N ₁	N ₀	E	R	S	F
---	----------------	----------------	----------------	----------------	----------------	---	---	---	---

2nd character of instruction words (HC = 0)

V	X ₇	X ₆	X ₅	X ₄	X ₃	X ₂	X ₁	X ₀	P
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	---

1st octet of additional instruction (HC = 1)

V	F ₈	F ₁	F ₀	K ₁	K ₀	E	R	S	P
---	----------------	----------------	----------------	----------------	----------------	---	---	---	---

2nd octet of additional instruction (HC = 0)

V	C ₇	C ₆	C ₅	C ₄	C ₃	C ₂	C ₁	C ₀	P
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	---

Delay character (HC = 0)

V	T ₇	T ₆	T ₅	T ₄	T ₃	T ₂	T ₁	T ₀	P
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	---

Service characters (HC = 0)

V	S ₇	S ₆	S ₅	S ₄	S ₃	S ₂	S ₁	S ₀	P
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	---

Data or status character (HC = 0)

V	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	P
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	---

VME	BE ₇	BE ₆	BE ₅	BE ₄	BE ₃	BE ₂	BE ₁	BE ₀
-----	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

II Transmit bus

Preselection acknowledgment
characters (echos)

V	N ₄	N ₃	N ₂	N ₁	N ₀	E ₂	E ₁	E ₀	P
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	---

Data or status characters

V	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	P
---	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	---

Plate 9 Bit positions of the various characters in the receive bus (BR) and the transmit bus (BE)

ANNEX E

COMMON STRUCTURE AND INTERFACES FOR

GUIDANCE AND CONTROL SYSTEMS WITH

DIGITAL DATA PROCESSING

**“Einheitliche Grundstruktur und Schnittstellen in Flugführungssystemen
mit digitaler Signal-verarbeitung”**

**(Common structure and interface for guidance and control systems
with digital data processing)**

Preliminary Specifications for a serial and a parallel data bus for digital guidance and control systems have been laid down by a group of experts from the aircraft and avionic's industry headed by the German Minister of Defence. The specifications are documented in:

A serial Real Time Data Bus

A parallel Real Time Data Bus

and given here.

CONTENTS

	Page
1. SERIAL REAL-TIME DATA BUS	E-3
1.1 Data Bus Architecture	E-3
1.2 Data Bus Operation	E-3
1.3 Data Code, Bit Synchronisation, Transmission Rate	E-3
1.4 Message Formats	E-3
1.5 Modes of Transfer	E-4
1.6 Electrical Specifications	E-5
2. ELECTRICAL INTERFACES AND DATA TRANSMISSION ON PARALLEL BUS SYSTEMS	E-11
2.1 Introduction	E-11
2.2 The Parallel Bus Concept	E-11
2.3 Basic Architecture for Parallel Bus Systems	E-11
3. A PARALLEL REAL-TIME DATA BUS	E-16
3.1 Overview	E-16
3.2 Bus Signals	E-17
3.3 Message Transfers	E-18
3.4 Electrical Specifications	E-19
3.5 Mechanical Specifications	E-21

1. SERIAL REAL-TIME DATA BUS

1.1 Data Bus Architecture

This specification describes a serial data bus for digital on-board data systems requiring rapid and comprehensive data transfers between distributed subsystems.

The serial bus system consists of the data bus line and its interface electronics (*terminals*) as shown in Figure 1, and links a number of arbitrary digital subsystems. In order to increase system reliability the bus line may be replicated.

According to their capabilities four classes of terminals are distinguished:

- Class 1: Passive terminals, receivers only.
- Class 2: Receivers and transmitters; transmitting ability restricted to replies on request.
- Class 3: Receivers and transmitters; ability to generate request signals.
- Class 4: Receiving and transmitting functions as in class 3 terminals; additional ability to take over bus control (master function).

A serial bus system may include up to 64 of the above mentioned terminals in any combination. Class 3 and 4 terminals are integrated into a priority chain.

1.2 Data Bus Operation

Transmission on the bus line functions asynchronously in a half-duplex time multiplex manner. In general the data transfer takes place in a command/response mode, except for global transfers as defined in 1.5.2. The information flow on the data bus is comprised of messages which are, in turn, formed by sections and sub-sections as described in 1.4.

Bus Control

The message transfers on the data bus may be controlled by any class 4 terminal. Only one bus controller may be active at a time, however. Bus control may be transferred to another class 4 terminal in two different ways: In the first way the active controller switches control to the selected terminal via a command message which in turn has to be acknowledged by the terminal taking over control. This direct transfer may also be requested by the terminal taking over via a request operation as described in 1.5.3. The second mode of transfer of control is an automatic priority-controlled transfer. In this case the present controller simply stops transmission, and the next controller takes over according to its rank in the priority chain of the terminals. This mechanism is particularly important in case of a failure of the bus controller and is described in more detail in par. 1.5.1.

1.3 Data Code, Bit Synchronization, Transmission Rate

The code used for data transmission is a pseudo-ternary bi-phase code as shown in Figure 2. A logic one is represented by a bipolar coded signal 0/1 (i.e., a negative pulse followed by a positive pulse). A logic zero is a bipolar coded signal 1/0 (i.e., a positive pulse followed by a negative pulse). A transition through zero occurs at the midpoint of each bit time. The absence of any pulse indicates a gap (no traffic on the line). Individual clock generators are provided in all terminals. Bit synchronization is achieved on the bus by extracting the clock signal from the bi-phase code during message transfers. That is, the terminal transmitting is clock master at the same time, and all other terminals are slaved to its clock.

Transmission rate on the bus is 1.0 Mbit/sec, long and short term stability are still to be defined.

1.4 Message Formats

A message is composed of a data section and a command/response section or of a command/response section only as shown in Figure 3. The data is transmitted first starting with the least significant bit (LSB).

1.4.1 Command/Response Section

The command/response section is comprised of a function part, an address part, a signation part, and a cyclic redundancy check (CRC) character for data protection.

Function

The function part comprises 5 bits which can be utilized freely for either command and control purposes or further subaddressing within the remote subsystem as dictated by the individual subsystem requirements.

Address

The next six bits contain the address identifying up to 64 addressable remote terminals. This address can be used in two modes, depending on the signation accompanying it: In the first mode, exactly one terminal is addressed. In the second mode one group or several groups of terminals are addressed. The assignment of different groups to addresses can be chosen freely according to individual needs. In this mode a certain address code in combination with a certain signation will thus result in addressing a user-specified group of terminals.

Signation

The following five bits of the signation part specify the type of the message. As shown in Figure 4, the most significant bit (K00) indicates whether the message represents a command or a response. (In case of a response the answering terminal will identify itself by inserting its own address into the address part.) A logic one indicates a response while a logic zero indicates a command to (a) remote terminal (s). The next bit is used to determine the address mode: A logic one indicates that one particular terminal is addressed while a logic zero indicates addressing of a group of terminals. The third bit contains information as to whether the message includes a data section or not. A logic one indicates a message with data whereas no data is transmitted when this bit is zero. The function of the last two bits (K 03, K 04) has not been defined yet and is left for future extensions.

Cyclic Redundancy Check

The last eight bits of a message are reserved for an error checking code. All messages must be protected. Error detection is obtained utilizing a cyclic code. The code polynomial has not been specified so far. However, all terminals sharing one common multiplexed data bus must use the same polynomial.

1.4.2 Data Section

The data section of a message is comprised of data grouped into bytes, and may amount to up to 29 bytes. The least significant bit of each byte shall be transmitted first (Fig.3). No further restrictions are imposed on the data section, i.e. its content may be interpreted by the remote terminals in any way (e.g. as status words, sub-addresses, control functions, and so on).

1.5 Modes of Transfer

1.5.1 Message Synchronization

Message synchronization on the data bus is done using gaps following the messages, as shown in Figure 5.

The end of a given message transmitted over the bus line is signaled to each terminal through the absence of any traffic and thus synchronization pulses. During the following gap free running oscillators in all terminals maintain terminal bit synchronization.

The gap consists of one bit-time of no bus activity at all (PZ 00) followed by two bit-times during which any class 2 or 3 terminal may issue a request signal as described in 1.5.3 (PZ 01/02). The following bit time within the gap is reserved for compensation of running time delays.

If the presently active controller maintains bus control the gap is terminated by the subsequent message starting with PZ 04. This may be a message transmitted by the bus controller, or a response from one of the terminals, or just a sync signal issued by the controller, consisting of a logic one. This signal is then interpreted as the end of a message followed by a new gap.

If the bus controller does not keep the control function (e.g., in case of a failure), it will not continue transmission at bit time PZ 04. Instead, the functioning terminal having highest priority will take over bus control. The priority time slot assigned to it within the gap. In this scheme, the highest-ranked terminal is assigned time slot PZ 05, the other terminals following in descending priority order. Accordingly, slot PZ 68 is assigned to the lowest priority terminal.

The terminal taking over control will do so by transmitting a status message starting from 'its' time slot. Following this message it will begin all further messages at PZ 04, thus indicating that it has assumed bus control.

1.5.2 Message Exchange

Individual Transfers

Individual transfers take place between the bus controller and a single terminal. Individual transfers always consist of a message from the controller and a response from one terminal. This transfer sequence may not be interrupted.

Data Transfer Controller-Terminal

The bus controller issues a message containing a data section and a command section. After message validation the addressed remote terminal answers by sending an acknowledgement message containing a response section only. This acknowledgement is issued following a gap of exactly four bit times, as indicated in Figure 6(a).

Command Transfer Controller-Terminal

The bus controller issues a message consisting of a command section only. Upon message receipt and validation the addressed remote terminal answers by sending an acknowledgement message containing a response section only. This acknowledgement is issued following a gap of exactly four bit times, as indicated in Figure 6(b).

Data Transfer Terminal-Controller

The bus controller issues a message consisting of a command section only. Upon message receipt and validation the addressed remote terminal answers by transmitting a message containing a data and a response section starting after a gap of exactly four bit times, as shown in Figure 6(c).

Global Transfers

During global transfers the bus controller addresses one or more groups of remote terminals using special addresses/signations as specified in 1.4.1. Global transfers or transfer sequences may not be interrupted.

Global Command

The controller issues a command message to a group or several groups of terminals. The remote terminals do not respond by an acknowledgement (Fig. 7(a)).

Global Read Operation

The controller issues a command message to a group or several groups of terminals. The addressed terminals answer by each sending a message containing a data section and a response section. These messages are separated by 4 bit-gaps, the transmission sequence being determined by the priority chain of the terminals: The addressed terminal having highest priority will transmit first, followed by the other terminals in descending priority order. No acknowledgement is given by the bus controller (Fig. 7(b)).

1.5.3 Request Operation (Interrupt)

Any remote terminal which is able to transmit, can request bus controller service by transmitting a positive voltage level during the double time interval PZ 01/02 of a gap. This request serves as a 'collective' interrupt since more than one terminal may generate a request signal at a given time.

After completion of the present bus operation the controller will respond to the request(s) by issuing a global read command which may be addressed to all terminals or to a group of terminals only. Each terminal addressed will now answer by placing a status message on the bus. Priority conflicts are resolved by an orderly transmission of status messages according to the terminal's rank in the priority chain, as described in 1.5.1.

Now the controller services all requesting terminals via individual transfer sequences as shown in 1.5.2.

1.6 Electrical Specifications

The serial data connects up to 64 addressable users with each other. It includes the bidirectional signal line and the transmitting/receiving and control circuits required for data traffic.

An elongated signal line is planned as the configuration. Other line configurations, for example, ring lines, are not excluded.

Each user must be connected to the signal line by means of a standard coupling. It must contain the required transmission/receiving and control circuits and must satisfy the following connection requirements. At the present time further details have not been specified.

1.6.1 Data Path Signal Transmission

Signal level

Positive logic is to be used on the data path. According to Type C ARINC 419, the following are specified as signal levels:

Transmission side	High	+ 4.5 to + 5.5 V
	Null	- 0.5 to + 0.5 V
	Low	- 4.5 to - 5.5 V
Receiving side	High	+ 3.5 to + 6.5 V
	Null	- 1.5 to + 1.5 V
	Low	- 3.5 to - 6.5 V

Transmission line

A shielded "Twisted Pair" or a coaxial cable is to be used as a transmission line, which is terminated at both ends by a characteristic impedance.

Characteristic impedance	$Z_L = 60\Omega \pm 10\%$
Line length	$< 200 \text{ m}$

User coupling

The connection of the users to the signal line is done by means of a transformer coupling according to Figure 8. Details of the receiver and transmitter connections remain to be worked out. It is recommended to structure the internal user interface in such a way that the expected changes in the bus technology will not influence the technology of the user's circuits.

1.6.2 Transmitter/Receiver Circuits

Not specified at present.

1.6.3 Signal Correspondence in Time

Depending on the signal transit time, transient time and switching times along the data path, message tacts and pause times are to be specified as follows:

$$\begin{aligned} PZ &= 2 \times \text{signal transit time} \\ BT &= T \end{aligned}$$

Remark:

The value $T = 1 \mu\text{s}$ is to be approached for 100 m of line length, i.e., a growth bit rate of 1 MBit/s.

The time unit T must be produced in each addressable user by its own oscillator. The oscillator accuracy must be specified between two message transmissions as a function of the maximum possible pause time. After the pause has elapsed, a maximum phase displacement of $\pm 20\% T$ can occur among users.

The message tact is determined by the transmitting user in each case. Upon reception, the tact of the user is to be synchronized with the received signal. The evaluation of the pause time between two message transmissions must occur after the synchronizing tact. In order to prevent a blocking of the data path by repeated disturbance transmissions of the user, a protective circuit is to be provided in each user for limiting the transmission time. This protection circuit is activated when a transmission has exceeded the maximum permissible message length of 256 bits and then it pulls it out of the traffic.

1.6.4 Voltage Supply

Each user has its own voltage supply and supplies the required energy corresponding to the prescribed signal level only during transmission to the bus. The individual users can be connected to various onboard networks.

Mechanical specifications

The mechanical specifications of the serial data path are not yet specified at the present time.

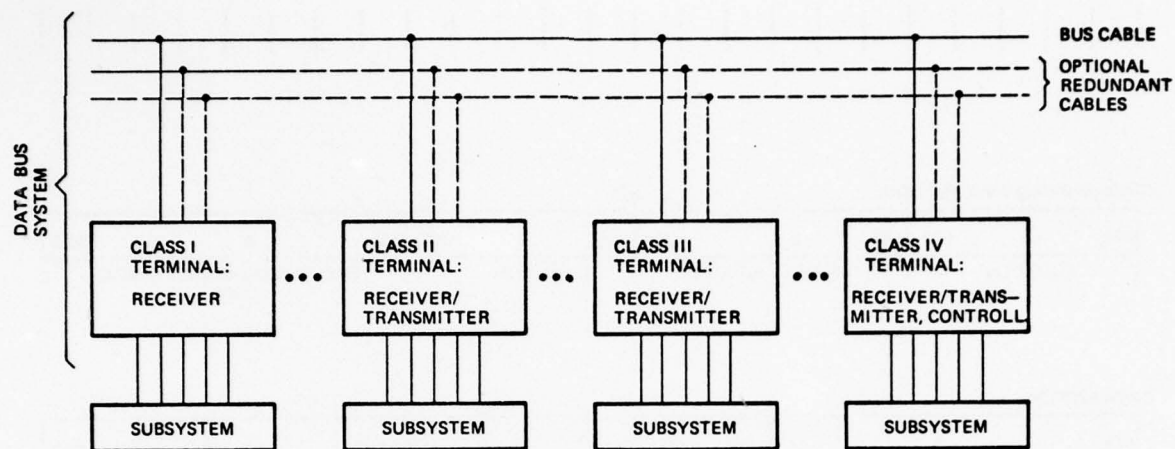


Fig.1 Architecture of the serial multiplex data bus

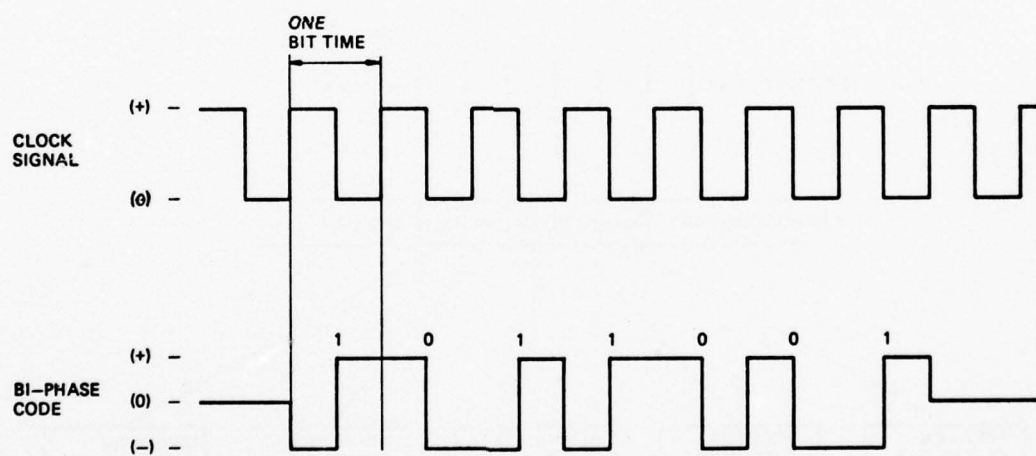
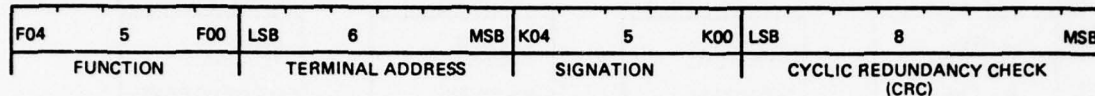


Fig.2 Pseudo-ternary bi-phase code

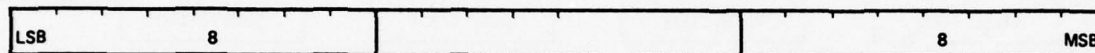
BIT TIMES:



COMMAND/RESPONSE SECTION:



DATA SECTION:



DATA: UP TO 29 BYTES

Fig.3 Message format

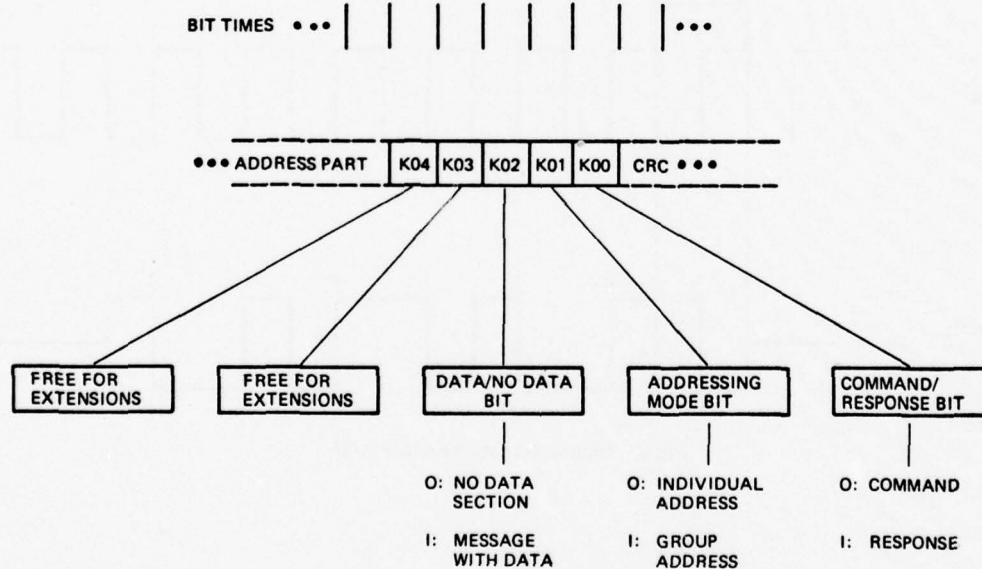


Fig.4 Signation part – bit assignment

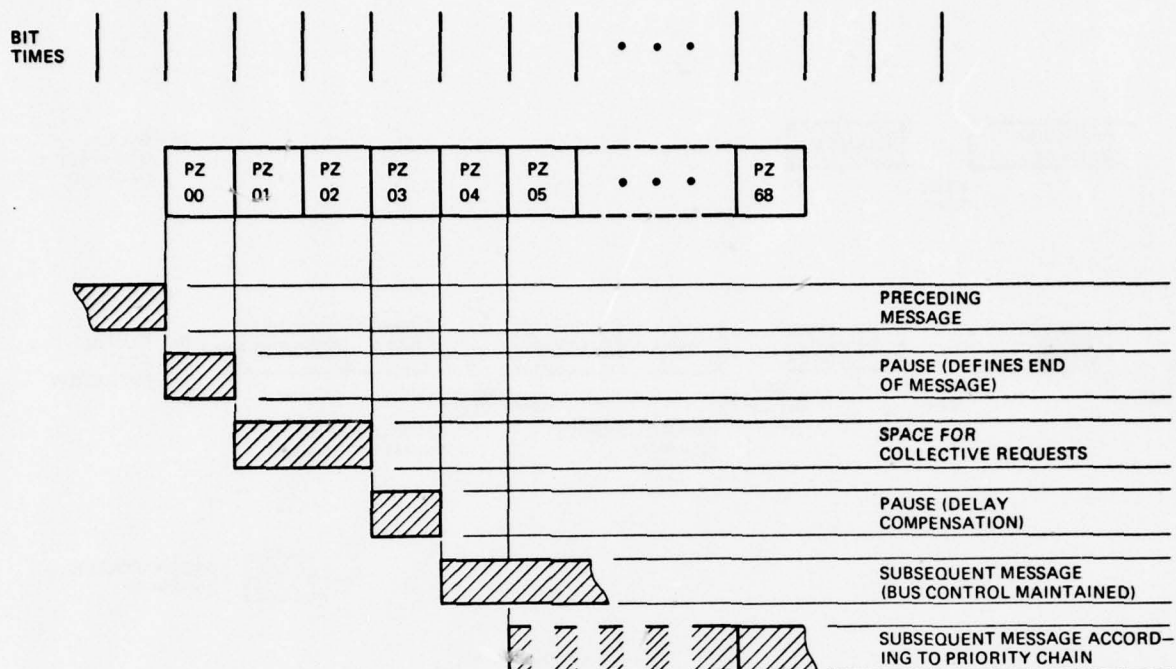


Fig.5 Gap structure

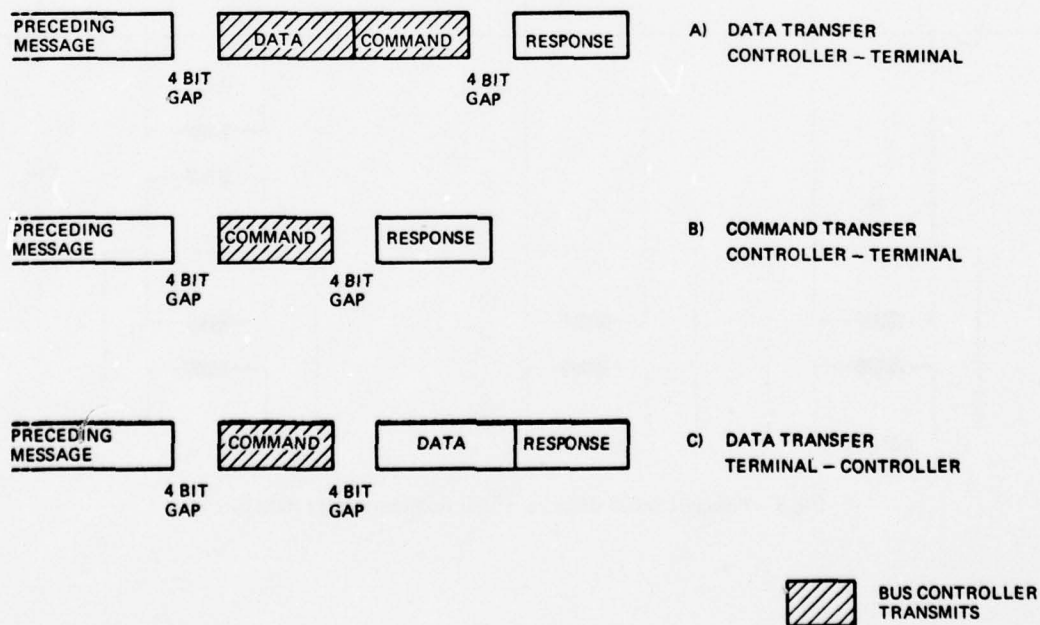


Fig.6 Individual transfer modes

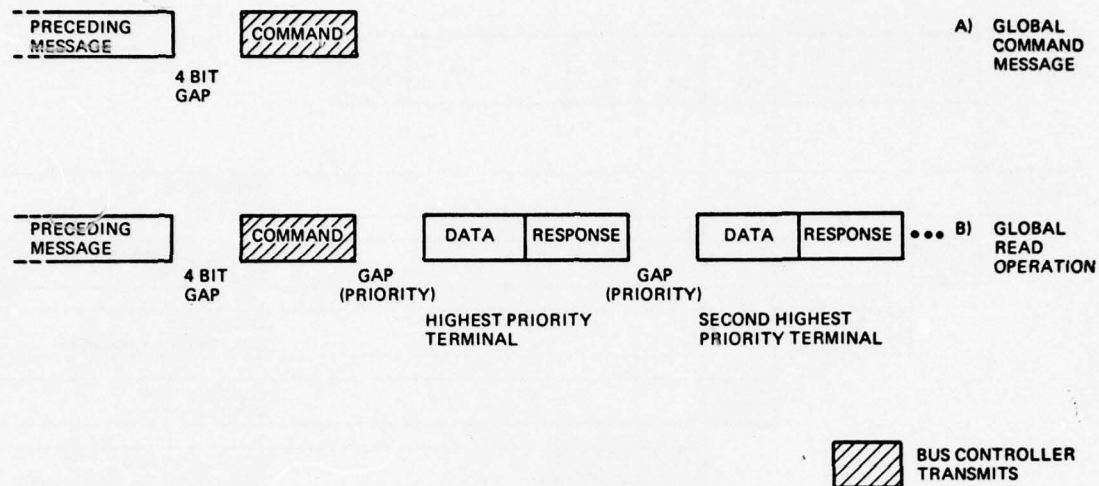


Fig.7 Global transfer modes

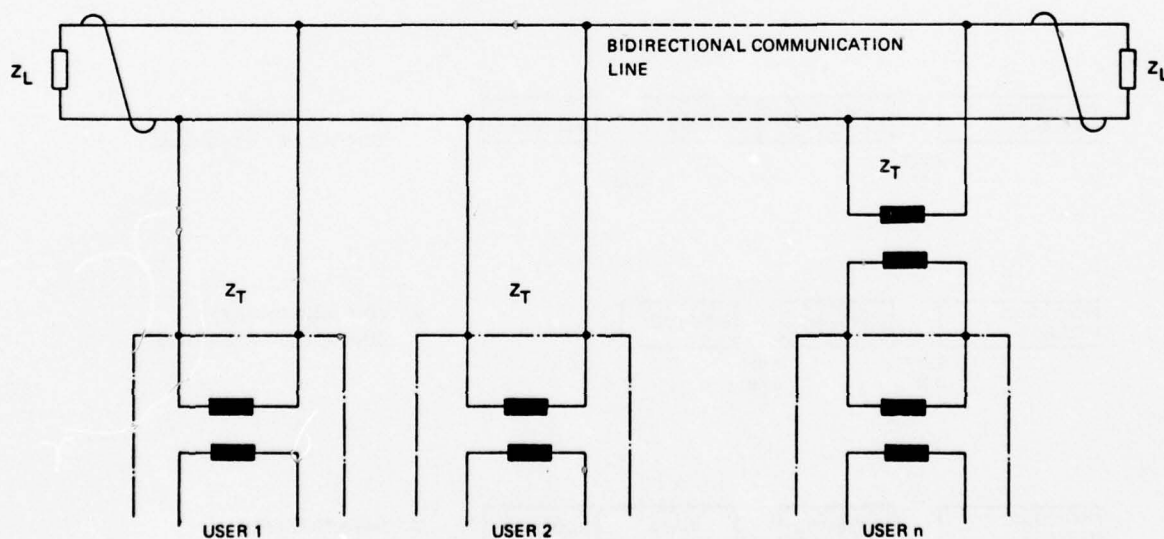


Fig.8 Principle block diagram of the serial multiplex data bus

2. ELECTRICAL INTERFACES AND DATA TRANSMISSION ON PARALLEL BUS SYSTEMS

Standardized Parallel Bus Systems have not been widely used in digital guidance and control systems. Therefore a short introduction to the general problem areas of parallel bus systems is given here in order to provide the reader with the tools of better understanding the following parallel bus specifications. This introduction is not part of the specifications.

2.1 Introduction

Functional partitioning of guidance and control systems can have high impact on cost, reliability, maintenance, and other factors.

If data processing systems are to be partitioned from the unit level (i.e. a box containing a number of modules) to the module level (e.g., multiplexers, A/D converters, memory modules . . .), standard electrical interfaces and standard transfer procedures must be established between the different modules.

Besides, the potential advantages of supplier independent line replaceable modules can be realized only if there is a large amount of replication of standard modules. Since digital signal processing modules can perform a variety of data acquisition and processing operations they seem especially well suited for standardization.

Investigations towards appropriate solutions for electrical interfaces and transmission techniques between a large number of similar or identical modules have resulted in the development of a parallel bus concept as the best compromise between flexibility, cost, performance, complexity, and adaptability to technical innovations. As will be shown below, the parallel bus concept is the appropriate solution for the provision of standard interface and transmission techniques between modules within one unit.

2.2 The Parallel Bus Concept

Contrary to the serial bus concept which is mainly intended to interconnect autonomous avionic subsystems with separate power supplies, the parallel bus is intended to interface modules within one physical unit. DC-coupling is therefore feasible or even required. The cable length of a parallel bus within one unit will usually not exceed about 50 cm. As opposed to the serial bus discussed in "Multiplexed Data Bus Technology" within this report, the problems of the bus as a transmission line, and the AC-transformer bus coupler do not dominate the parallel bus concept. The choice between a serial and a parallel bus to interface modules within one unit must, among other considerations concerning speed, reliability, weight, power consumption, also be based on the logic required to control the bus and bus operation. Figures 1 and 2 demonstrate the principal logic needed for serial data transmission and reception. With the assumption that in general the information (address, functions and data) is generated in parallel coded form and received in parallel coded form in the receiving module, the logic circuitry is considerably less complex if a parallel bus transfer concept is selected as demonstrated in Figure 3. As almost all present processors or on-board computers are using a 16 bit data structure, an appropriate bus solution will be the use of 16 parallel data lines. The number of address lines and function lines will be dictated by the equipment requirements. In general five address lines and three to five function lines should be sufficient for guidance and control equipment. As compared to a serial bus, the parallel bus concept obviously requires a higher number of interfacing lines between the different modules.

Since printed circuit techniques may be employed, however, and harnesses are not required, wiring does not present any problem here. Besides, the use of bidirectional data lines offers solutions to keep the number of wires in reasonable limits. The logic circuitry is somewhat higher compared to unidirectional data bus lines. The number of wires and pins of the plugs and sockets however is considerably less.

2.3 Basic Architecture for Parallel Bus Systems

2.3.1 General Configuration

A basic parallel bus system consists of the bus lines, and interface electronics to link a number of arbitrary digital modules to the lines. According to the capabilities of these modules, several classes can be distinguished:

- Class 1: Passive modules (receivers only, e.g. digital to analog converters)
- Class 2: Active modules (transmitters only, e.g. analog to digital converters)
- Class 3: Modules which can be active or passive (receivers and transmitters, e.g. random access memory modules)
- Class 4: Transmitting modules with the additional ability to generate request signals.

The data transmission on the parallel data bus must be controlled by a bus controller. The number of modules linked to a parallel bus system is limited due to the maximum length of the bus. The bus length is in turn limited by the amount of energy required to drive the bus as well as by the delays occurring on the bus lines. Delays exceeding one bit time result in transmission errors. With present technology not more than 32 functions may be connected to one parallel bus.

Figure 4 depicts an example of a parallel bus system. Concepts are known having non-dedicated bus lines only as well as concepts with dedicated address and request lines. Dedicated lines result in more wiring as compared to bus lines. However, the interface circuitry is less complex, and data transmission is not as much subject to errors which is particularly important for the addressing part. In most cases one unit (box) does not contain more than 32 modules. In this case, Figure 4 would be the block diagram of a complete subsystem. In equipment like this, the bus controller and the control processor are combined within one module as a rule. In this case the configuration functions as an autonomous subsystem.

2.3.2 Data Bus Hierarchy

Complex equipment for guidance and control can comprise more than one parallel bus system if more functional units and elaborate on-board digital computers are required. As a rule, the on-board computer provides for control data processing within the equipment. The link between the different bus controllers is provided by a system controller (Fig.5). In general, the parallel bus is called "dataway" and the bus linking the different bus controllers to the system controller "data highway". The data highway must provide for data transmission over a distance of up to a few meters because it must link several parallel dataway systems, each of which is usually housed in one ATR box. Due to different bus lengths and requirements, the dataway and the data highway may have to be of a different nature. If high data rates are required the data highway must be a parallel multiplex bus system as well. In other cases, a serial multiplex data bus may be sufficient. Apparently within guidance and control, system configurations as shown in Figure 5 on the equipment level are not implemented often, because no attempt towards standard specifications of the data highway could be located in the avionics field, as compared to the CAMAC system for large industrial and nuclear acquisition systems.

2.3.3 Hierarchical Serial/Parallel Bus System Configurations

Digital equipment, consisting of several parallel bus systems, may be integrated into one system via serial multiplexed bus lines in two different ways, depending if the equipment shall function autonomously or not.

In equipment which functions autonomously and where modules must not be addressed under the supervision of the serial bus controller, parallel bus systems can be coupled to the serial bus by an adapter module which, at the same time, is one of the parallel bus modules (Fig.6).

In equipment where modules of the parallel bus system must be addressed by the master bus controller directly and where the subsystems shall not operate autonomously, the system architecture of Figure 5 is an appropriate solution. The data highway must be implemented as a serial bus system in this case and, the system controller must function as a serial bus controller.

2.3.4 Replicated Parallel Bus Systems

Reliability requirements may dictate the use of redundancy for parallel bus systems. Parallel bus systems can be architected with full or partial redundancy. Figure 7 depicts one example.

Here the processors and the bus controllers as well as the modules having critical functions are duplicated. Modules not essential for the mission requirements are provided only once in one of the two parallel bus systems. If one of the duplicated critical modules fails, it will be switched off, and the system automatically addresses the stand-by module.

If one of the processors fails, the remaining one will perform all data processing operations required for the mission.

Various modes of operation are feasible, as dictated by the individual system requirements. In normal operation one of the systems can be switched off or both systems can operate synchronously with one of the bus controllers being master. Other modes could be different tasks for the two systems in normal operation, and emergency operation for the remaining system in case of failures.

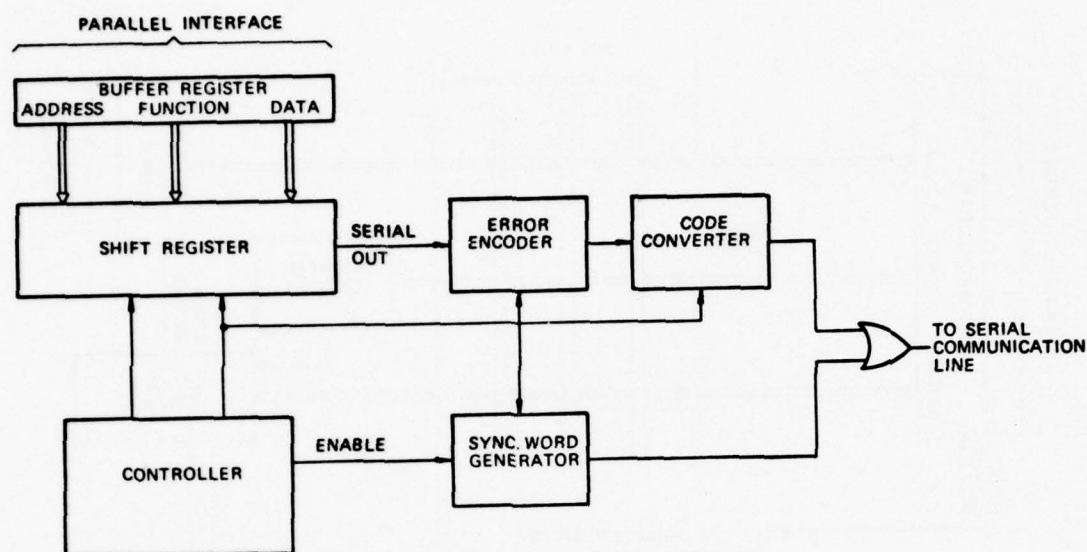


Fig. 1 Serial data transfer: transmitter logic (simplified)

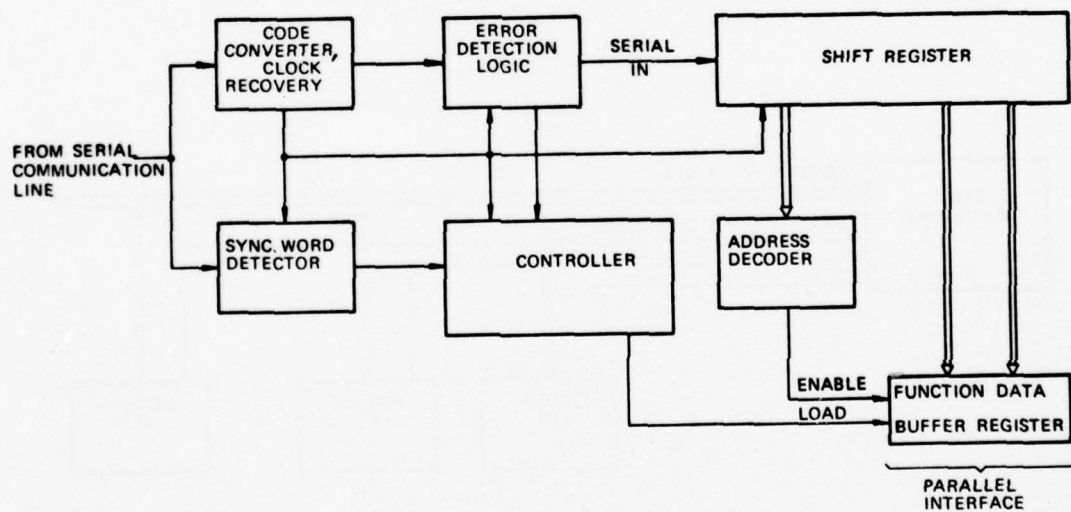


Fig. 2 Serial data transfer: receiver logic (simplified)

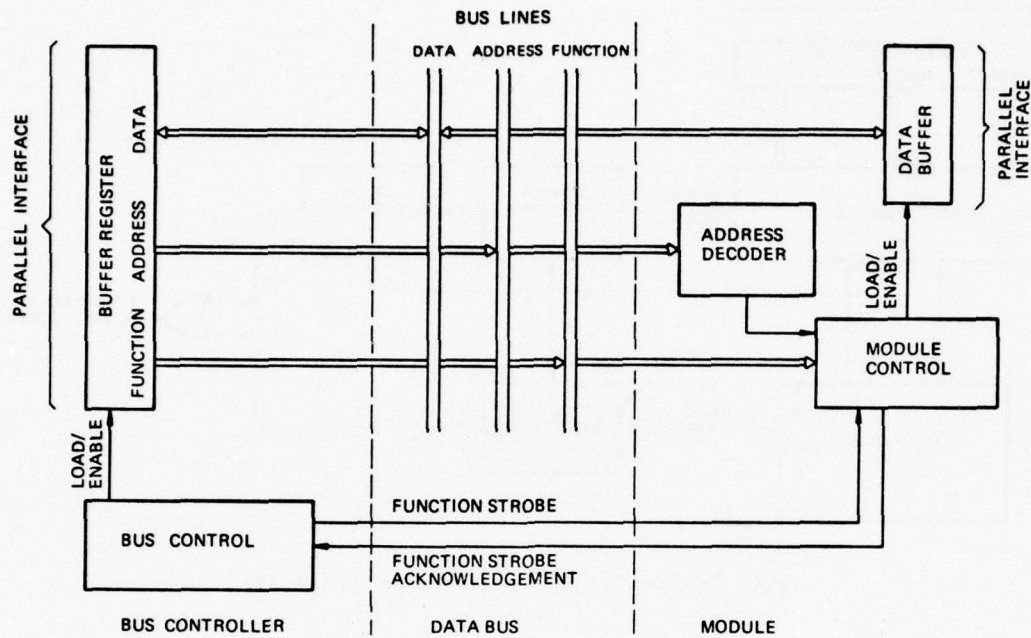


Fig.3 Parallel data transfer: logic units (simplified)

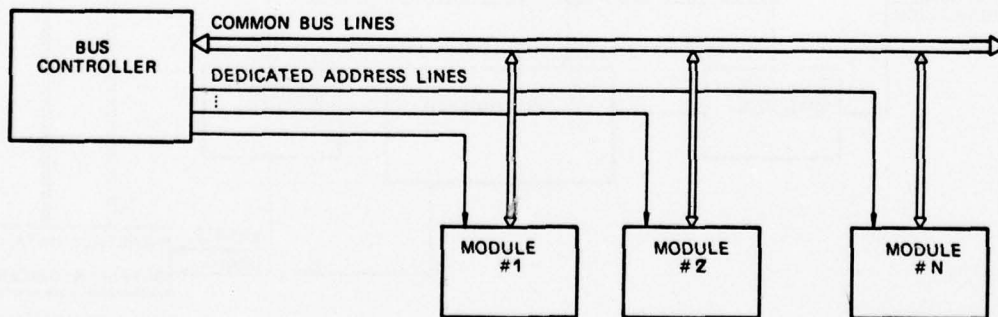


Fig.4 Basic parallel data bus configuration

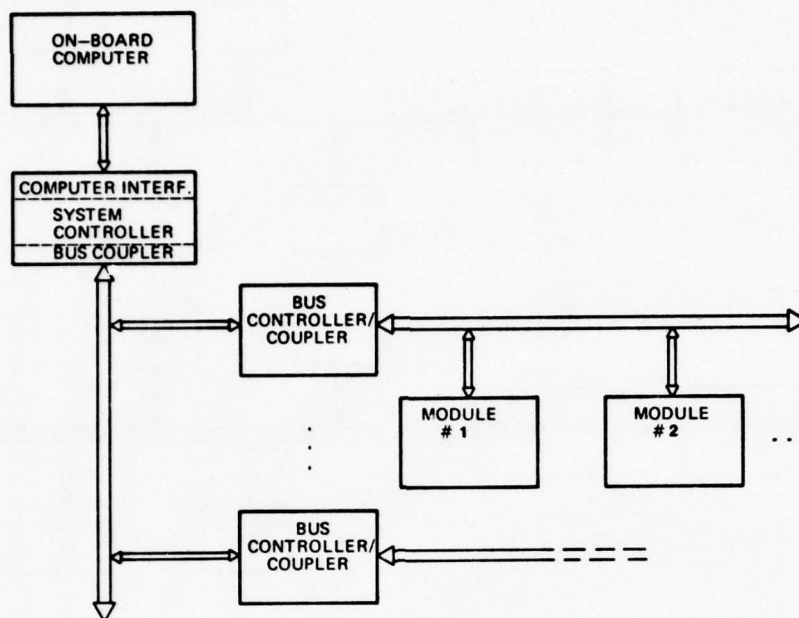


Fig.5 Parallel bus system hierarchy

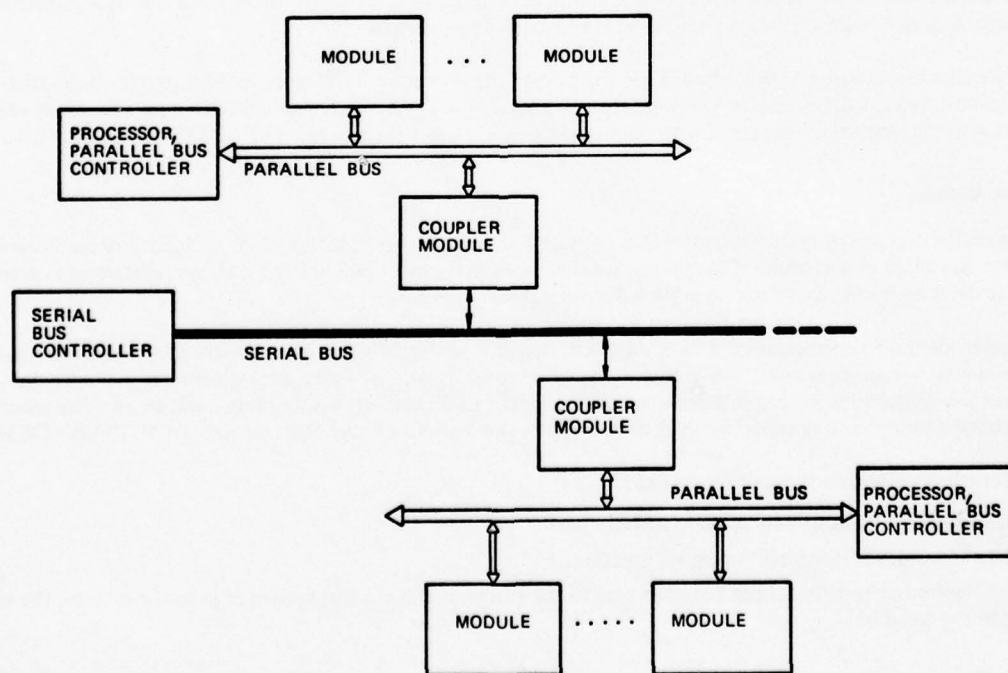


Fig.6 Parallel/serial bus system configuration

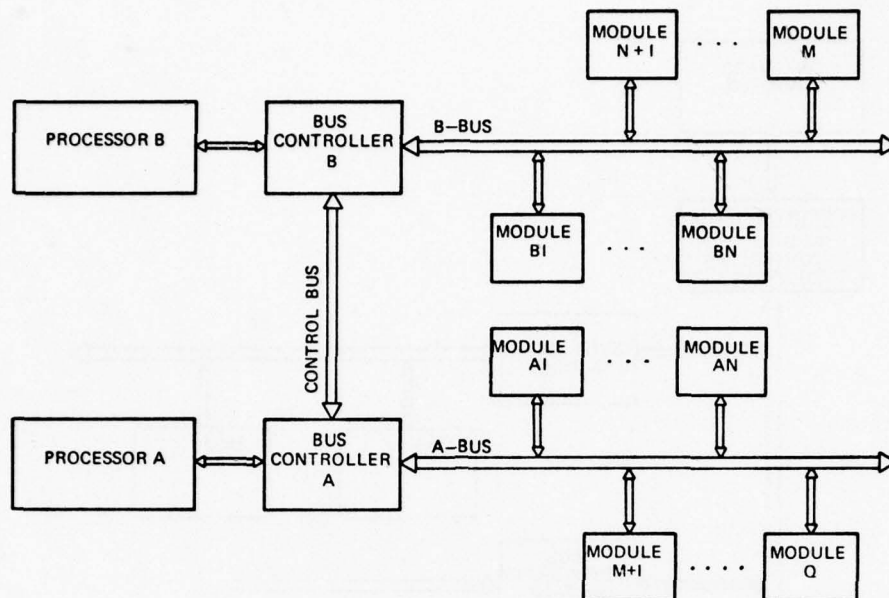


Fig. 7 Partially duplicated bus system

3. A PARALLEL REAL TIME DATA BUS (Preliminary German Specifications)

3.1 Overview

This specification describes a parallel bus for digital on-board data systems requiring rapid real time data transfers between digital devices within guidance and control and avionics equipment.

The parallel bus system consists of bus lines and interface electronics. It links up to 32 digital modules such as A/D converters, multiplexers, digital inputs/outputs, etc. The parallel bus system shall be controlled by a bus controller (see Figure 4 in the preceding chapter "Electrical Interfaces and Data Transmission on Parallel Bus Systems").

3.1.1 The Modules

Each device of a digital system which shall be serviced via the parallel bus, and which includes standard bus interface electronics is specified as a module. The system designer can specify as a module any function or group of functions required, as dictated by the individual operational system requirements.

However, the bulk of functions can be included in two groups. The first group consists of input/output modules linking the digital system to sensors, actuators, displays, and controls such as digital-analog converters, analog-digital converters, synchro-digital converters, multiplexers, digital inputs and digital outputs, discretes, and so on. The second group includes digital data processing functions such as processors and different kinds of memories (RAM, PROM, ROM).

Each module comprises three different parts:

- The interface electronics to the parallel bus,
- The electronics performing the digital function,
- The electronics to connect the function part to the guidance and control system (e.g. to the sensors, the actuators, or to the serial bus).

3.1.2 The Bus Controller

All data transfers are routed by a bus controller. The controller generates all necessary control signals as dictated by the timing of the transfers. Transfers can be initiated by the controller, or by a module via an interrupt signal. (Normally the bus controller and the processor are one unit.)

3.1.3 Data Bus Operation

Transmission on the parallel bus functions asynchronously in a half duplex manner. In general the data transfer takes place in a command/acknowledge (hand-shake) mode. Data transmission on the bus is restricted in transfers from and to the bus controller. No provisions are made for direct module to module transfers via the parallel bus. (If direct module to module transfer is required, dedicated data lines must be provided outside the bus.)

TABLE I

Bus Signals

<i>Section</i>	<i>Signal Name</i>	<i>Symbol</i>	<i>Line Characteristic</i>	<i>Signal Direction</i>
Command	Module Address	N_i ($i = 1 \dots 32$)	Dedicated Lines	Controller → Module
	Module Function	F00, F01, F02	Unidirectional Bus Line	Controller → Modules
	Direction of Transfer	RW	Unidirectional Bus Line	Controller → Modules
	Reset	RST	Unidirectional Bus Line	Controller → Modules
Data	Data	D00 – D15	Bidirectional Bus Line	Controller ↔ Modules
Control	Function Strobe	FS	Unidirectional Bus Line	Controller → Modules
	Function Strobe Acknowledgement	FSA	Unidirectional Bus Line	Modules → Controller
	Collective Alarm	ALI	Collective Line	Modules → Controller
	Alarm Interrogation	ALA	Unidirectional Bus Line	Controller → Modules
Special		NSL1, NSL2 SL1, SL2	Dedicated Lines Unidirectional Bus Line	Controller → Module Controller → Modules

3.2 Bus Signals (see Table I and Figure 1)

The bus signals can be grouped into a command section, a data section, a control section, and a special purpose section.

3.2.1 Command Section

The command section is comprised of an address part, a function part, and a transfer direction part.

Address

The address part consists of 5 bits to address up to 32 modules. Address decoding, however, is done centralized in the bus controller and not decentralized in the modules. Therefore the modules must be addressed by the bus controller via dedicated lines.

Function

The function part (F00–F02) comprises 3 bits which are decoded decentralized in the module. These bits can be utilized freely for either command and control purposes, or for further subaddressing within a module as dictated by the individual subsystem requirements. F00 is the most significant bit.

Transfer direction

The signal on the Read/Write bus line determines the transfer direction. A data transfer from any one module to the parallel bus controller (Read Operation) shall be coded as a logic 1 while a transfer from the bus controller to any module (Write Operation) shall be coded as a logic 0.

Reset

The Reset signal is used by the bus controller to set all modules to a defined initial state, when raised to a logic one.

3.2.2 Data Section

The data section is comprised of 16 bits. D00 is the most significant bit. The 16 data lines are bidirectional bus lines.

3.2.3 Control Section

Transfer control strobes

The strobe signal FS and the strobe acknowledgement signal FSA are used for the handshake operation during a message transfer. By means of a logic 1 on the strobe line FS, the module is to perform the command issued to the addressed module. A logic 1 on the FSA line shall indicate that the module has completed the required operation. The FSA Signal can be generated freely within a module as dictated by the module requirements. The pause between the transmission of the FS strobe signal and the reception of the FSA acknowledgement signal is interpreted as "Bus Busy Period".

Alarm control strobes

The collective alarm (ALI) is a signal that has rippled through the priority chain of the modules before reaching the bus controller, and indicates an alarm transfer request issued by any one module.

The alarm interrogation signal (ALA) is a command issued by the controller which causes the requesting module to identify itself by consecutively placing its signation on the data bus, as described in 3.3.

3.2.4 Special Line

Special dedicated lines

Two lines (NSL1, 2) are provided which can be used freely as dedicated lines to any modules. (Recommendation: The two lines shall be used in context with the address part. Any module connected to one of the two dedicated lines shall receive if a logic 1 is on the line, regardless of which module is addressed by the address part.)

Special bus lines

Two special bus lines (SLI1, 2) are provided which can be utilized freely as dictated by the requirements. (Recommendation: One of the special bus lines shall be used for transmission of a parity bit if internal data protection is to be provided.)

3.3 Message Transfers

There shall be three modes of message transfers:

- read transfer
- write transfer
- alarm transfer

In the read transfer mode data is transmitted from a module to the bus controller. In the write transfer mode one or more modules are receiving data issued by the bus controller. An alarm transfer is initiated by the bus controller to identify the interrupt source after a request has been issued.

3.3.1 Write Transfer (Fig.2)

The bus controller initiates a write transfer by issuing a decoded module address, and placing the function bits as well as the data on the bus lines. At the same time the R/W-line is put to a logic zero. After a delay of $> 0.2 \mu\text{sec}$, the controller activates the function strobe (FS) line thus indicating the validity of data on the bus lines. After taking over the data part, the addressed module answers by activating the function strobe acknowledgement (FSA) line for at least $0.6 \mu\text{sec}$. This in turn causes the controller to reset the FS line to a logic zero. Recognizing the trailing edge of the FS strobe, the module will reset its FSA signal and in this way terminate the write transfer. Function, data, and R/W-lines are to remain stable for at least $0.2 \mu\text{sec}$ after the reset of the FSA strobe.

3.3.2 Read Transfer (Fig.3)

The controller initiates a read operation by issuing a decoded module address, placing the function part on the respective bus lines, and setting the Read/Write line to a logic one. After a delay of $> 0.2 \mu\text{sec}$ the controller activates the function strobe line (FS), and in this way validates the read command. Recognizing this, the addressed module will place

its data on the data lines and raise the function strobe acknowledgement (FSA) line to a logic one. The controller will now store the information on the data lines, and confirm the take-over by resetting the FS line after a total delay of at least $0.6 \mu\text{sec}$. The module will now in turn reset the FSA line. Data and command bits are to remain stable for another $0.2 \mu\text{sec}$ after completion of the transfer.

3.3.3 Alarm Transfer (Fig.4)

Any module integrated into the module priority chain may issue a Collective Alarm signal. This signal is communicated to the bus controller via the ALI line. If two or more modules compete for the attention of the bus controller, the one having highest priority is selected for an alarm transfer.

After completion of its current operation, the controller will now respond by activating the ALA line, thus transmitting an interrogation signal to all modules. After a waiting period of $0.2 \mu\text{sec}$ plus a running-time period depending on the number of modules in the priority chain, the controller will raise the FS line. The module selected for transmission by its priority rank will now reply by activating the FSA line and simultaneously placing an alarm message on the data lines. This alarm message must contain the module's signiation thus enabling the controller to identify the transmitting module.

After taking over these data, the controller will reset the FS line to a logic zero which in turn causes the module to remove the FSA signal. Data has to remain stable, however, at least for another $0.2 \mu\text{sec}$. After this period, the module must have cancelled its ALI signal thus freeing the line for eventually following requests by another terminal.

3.4 Electrical Specifications

The data path connects a control unit with a maximum of 32 modules. It includes the signal paths and the transmission/receiving and the control circuits required for the data transfer. The circuits of the data path are to be made using CMOS technology, especially considering space applications which require circuits with increased protection against disturbances and small power losses.

The individual users of the data path must conform to the data path circuits discussed in the following, in which the signal level, logic and signal correspondence in time (timing) are specified. In addition, different logic, formats, levels and technology can be used within the individual users.

3.4.1 Data Path Signals

Signal level

Positive logic is required on the data path. The supply voltage of the data path must be $U_{VD} = 10 \text{ V}$. The following signal levels must be maintained:

- logical "0": $0 \text{ V} \leq U_{SL} \leq 2.5 \text{ V}$
(low state)
- logical "1": $7.5 \text{ V} \leq U_{SH} \leq 10 \text{ V}$
(high state)

Remark:

The data path can also be operated with a supply voltage of $5 \text{ V} \leq U_{VD} \leq 15 \text{ V}$. In this case no voltages can be taken off from this supply voltage for the user's circuits (see Section 3.4.4).

Signal lines

The signals specified in Table I must be transmitted on the data paths or the signal lines must be available. Specifically these are the following:

- 32 (maximum) address branch connections
- 8 unidirectional lines
- 16 bidirectional lines
- 1 request collection line

The following must be available as special lines which are used for special operations:

- 2 special address branch connections
- 2 free bus lines

3.4.2 Data Path

The transmission/receiving and the control circuits are prescribed for the signal transmission on the data path. However, they must be present in each user. They can be coupled with the user circuits within the modules.

CMOS-Tri-State transmitters are to be used as transmitters for transmitting data (D00–D15) and the module verification signals (FSA). The circuit characteristics must correspond to the building block CD 4043*. CMOS buffers are to be used as transmitters for transmitting other signals, and the circuit characteristics must correspond to the building block type CD 4049. The buffer transmitters for the branch connections represent an exception, and for them the circuit characteristic type CD 4028 is allowable. All CMOS inputs with an input capacity of $F_{in} = 1$ and 5 pF are allowable as signal receivers. The inputs for the address branch lines represent an exception, which can be operated at a maximum of $F_{in} = 3$.

All signal lines are to be connected against GND in the control unit through 10 k Ω resistance. The request collection line is an exception, which is to be closed off in the module by 10 k Ω with respect to GND.

Figure 5 shows the transmitter/receiver configurations for the individual signal lines which provide approximately uniform load states and approximately the same signal transit times.

Interface to the control unit (bus controller)

The interface to the control unit (bus controller) must be structured according to the functional and electrical specifications of the parallel data bus by appropriately designing the circuit within the bus controller. Figure 6 shows one of the possible logical circuit connections.

Interface to the module

The interface to the module must be structured according to the functional and electrical specifications of the parallel data bus within the circuit of each module. Figure 7 shows one of the possible logical circuit connections.

3.4.3 Signal Correspondence in Time

The duration of the individual parallel data bus operations (writing, reading and request operations) can be arbitrarily long and can be specified by the user according to his requirements. Depending on the signal transit time and switching times along the parallel data bus, only a minimum duration is specified.

In order to characterize a signal correspondence in time during an operation, the time unit T is used as a basis. It is specified depending on the reaction time of each parallel data bus. During a data bus operation, the following signal correspondences must be maintained:

- (1) The control unit (bus controller) can send the next strobe FS 1 T after the last parallel data bus operation at the earliest.
- (2) The strobe FS must be presented on the parallel data bus for at least 2 T .
- (3) The control unit (bus controller) can start the next parallel bus operation 2 T after turning off the strobe FS at the earliest.

The specified minimum duration of a parallel data bus operation amounts to 5 T (for example, 0.31 MHz operation or transfer frequency, respectively, for a parallel data bus length of about 0.5 m and a time unit of $T = 0.65 \mu\text{sec}$). The total duration of the parallel data bus operation depends on the modules which must first send a strobe response FSA before the control unit (bus controller) turns off the strobe FS. The strobe response FSA must be present at least for one time unit T . For the case that the strobe response FSA is absent, the waiting time of the strobe FS must be limited in the control unit (bus controller). The waiting time can be specified according to requirements.

The correspondence of the other parallel data bus signals is specified depending on the strobe FS:

- (4) The bus controller must transmit the commands and, if necessary, data or a request interrogation ALA 1 T at the earliest but 0.2 μsec at the latest before transmitting the strobe FS.
- (5) The bus controller must turn off the command and if necessary the data or the request interrogation ALA again 1 T after turning off the strobe FS at the earliest.
- (6) The interrogated module must turn off its strobe response FSA or its data and its module request ALO when turning off the strobe FS at the earliest and 1 T before the end of operations at the latest.

During the parallel data bus operation, it is not permissible to change either the data present or the priorities in a collection request.

Each parallel data bus operation ends with a data path release with the duration of 1 T , within which the control unit can recognize a collection request.

* CMOS building block types of RCA.

3.4.4 Supply Line

The following supply lines of the parallel data bus cabling are to be provided for the voltage supply of the data path:

P10V1/P10V2	2 lines	$U_{VD} = 10 \text{ V}$ CMOS-supply
GND1-GND3	3 lines	$U_{SS} = 0 \text{ V}$ ground
M7V	1 line	$U_{LL} = -7 \text{ V}$ PMOS supply
P5V	1 line	$U_{CC} = 5 \text{ V}$ TTL supply
and		
P15V	1 line	+ 15 V analog supply
M15V	1 line	- 15 V analog supply
HQ	1 line	High quality ground for analog components

Other voltages can be selected in special versions. However, they cannot influence the information exchange along the signal lines. Special module supplies with, for example, 400 Hz voltages must be directed through the peripheral wiring.

No turn-on sequence of the supply voltages can be supplied.

The loss lines should not exceed 1 Watt/cm frame length as a rule.

3.5 Mechanical Specifications

Completely compatible frames and insert units can be developed from the required dimensions, which are specified in Figures 8–12. The dimensions specify the installation of aviation boxes corresponding to type 1/2 ATR ARINC 404A, independent of the type of installation.

The individual insert units can consist of one of several socket boards. Uniform board lines must be present at each board insert point of the frame and two connectors must be available:

- one parallel data bus connector with fixed pin assignment, which provides coupling to the data path.
- a periphery connector, which makes it possible to have free internal wiring and the peripheral external connection.

In the frame, there must be a space for a divergent and parallel data path wiring and a periphery wiring.

One insert unit corresponds to one user unit. The sequence of the units on the frame is arbitrary and is only specified after installation of the branch connections of the parallel data bus. The bus controller as a rule should be arranged at the beginning or end of a frame. It can be connected with a computer or a higher order bus through its peripheral connector.

3.5.1 Frame and Housing

The construction design of the housing is arbitrary. The prescribed dimensions only affect the frame and a complete mechanical exchange capability must be provided for the insert units.

Figures 8 and 9 show the prescribed socket board lines and arrangement for an ARINC 1/2 ATR housing. The socket board must be in the transverse format for this application.

Board configuration

The socket boards must be directed through the board holders and the connector pins and coding pins. The construction details of the card holders as well as the stops for the boards is arbitrary. It is only necessary to maintain the holder tolerances given in Figures 9 and 10.

Board separation

The board separation is to be specified according to a raster scale of $n \times 1/4''$ (6.35 mm) depending on the required component height.

The components should be located preferably on the front side of the board. A minimum separation according to the lines given in Figure 9 must be maintained between two boards.

3.5.2 Socket Boards and Insert Units

Each socket board must have a parallel data bus bar and a periphery socket bar in the direction of the front side along the lower edge.

One insert unit which covers several socket boards cannot be designed using the stacking construction method. The

internal connection of the socket board of one insert unit must be done using the periphery connector, for example, with flexible bands or conduction plates in the frame.

The prescribed dimensions of the socket boards are shown in Figure 10. The board thickness outside of the free component area must be 1.5 ± 0.1 mm. The board edges must be deburred. The base material must have a GE quality of at least MIL-P-13949*. The adhesion material (layout) must at least correspond to MIL-P-55110 B-2.

Connectors according to the standard UTE 93-424, Type HE 801, are prescribed for the connector connections.

Parallel data bus connectors

A 41-pole connector, Type HE 801 E* 41S1 is to be used as parallel data bus connector according to Figure 11. The pin assignment for the parallel data bus connector is specified as absolutely necessary in Table II. Special contacts – for example, coaxial contacts – are not permissible.

Periphery connectors

Type HE 801 F** 53 Y connectors according to Figure 12 are to be used as periphery connectors. The pin assignment is arbitrary for internal wiring and peripheral external connections.

Remark:

Special versions with coaxial inserts are allowable.

3.5.3 Wiring

The wiring is to be connected to the frame in the special space assigned to it.

Parallel data bus wiring

The parallel data bus wiring is to be designed in a divergent manner according to the pin assignment given in Table II.

The bus line and the bus supply lines are to be connected with the corresponding connector contacts of all board insert points. The branch connections and the module request collection lines are to be connected as a function of the configuration and the priority sequence of the units.

Periphery wiring

The periphery wiring, for example, can be done using flexible bands, conduction plates or using conventional wiring methods.

3.5.4 Supply Unit

It is permissible to install a unit for voltage supply of the parallel data bus either inside or outside of the housing along the back side.

* MIL-P-13949 Plastic Sheet, Laminated, Copper-Clad For Printed Wiring Boards
 MIL-P-55110 B-2 Printed Wiring Boards
 UTE 93-424 Multi-contact connectors, Union Technology of Electricity
 ** Type of connector attachment

TABLE II
Parallel Data Bus Connector Assignment

41	GND	USS = 0 V	Ground line
40	P10V1	UVD = 10 V	CMOS-supply Special line
38	D15 LSB	(least significant bit)	
37	14		
36	13		
35	12		
34	11		
33	10		
32	09		
31	08	Data lines	
30	07		
29	06		
28	05		
27	04		
26	03		
25	02		
24	01		
23	D00 MSB	(most significant bit)	
22	GND	USS = 0 V	Ground line
21	ALA	Response to collection request	
20	ALO	Request output	
19	ALI	Request input	
18	SL2		Special line
17	RST		Reset
16	FSA		Strobe response
15	FS		Function strobe
14	RW		Transfer direction
13	F02 LSB	Command	
12	01	lines	Module function
11	F00 MSB		
10	NSL1		Special address line
9	NSL2		
8	N	Module address input	
7	P10V2	UVD = 10 V	CMOS-supply
6	P5V	UCC = +5 V	TTL supply
5	M7V	ULL = -7 V	PMOS supply
4	GND	USS = 0 V	Ground line
3	M15V	= -15 V	Analog supply
2	P15V	= +15 V	Analog supply
1	HQ	High quality ground for analog	
Pin. No.	Signal symbol	Definition	

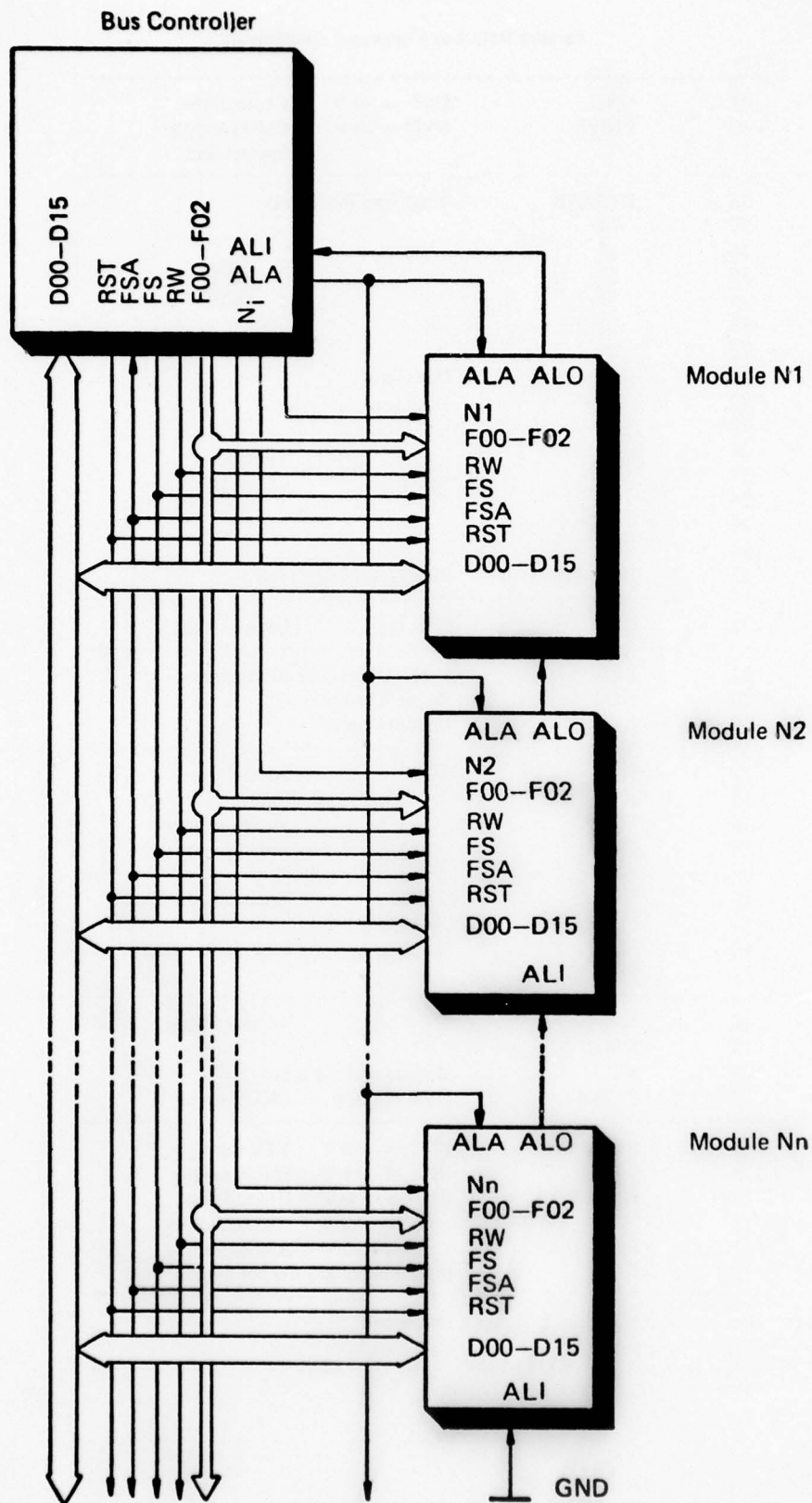


Fig.1 Block diagram of the parallel bus system

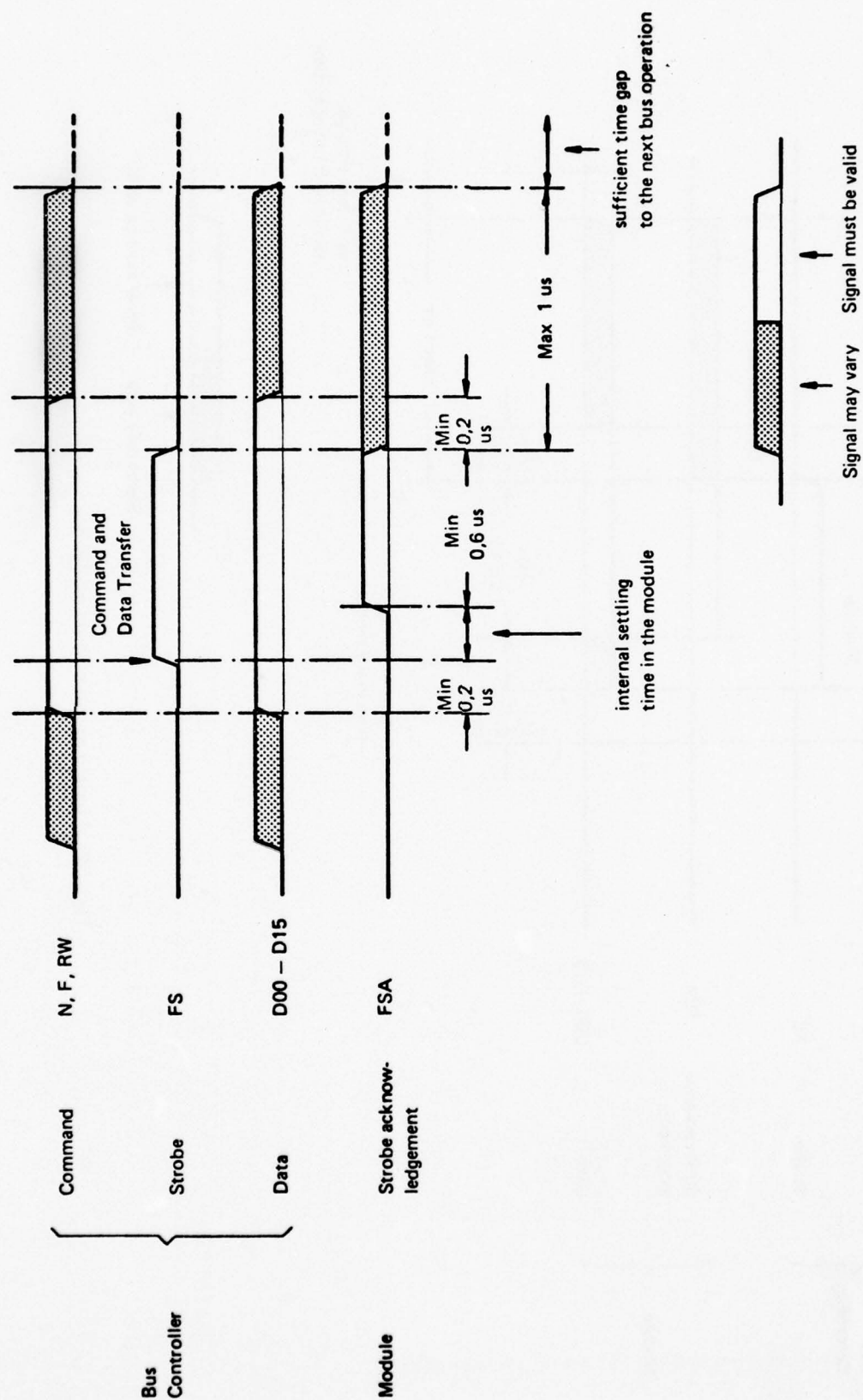


Fig.2 Write transfer operation on the parallel bus

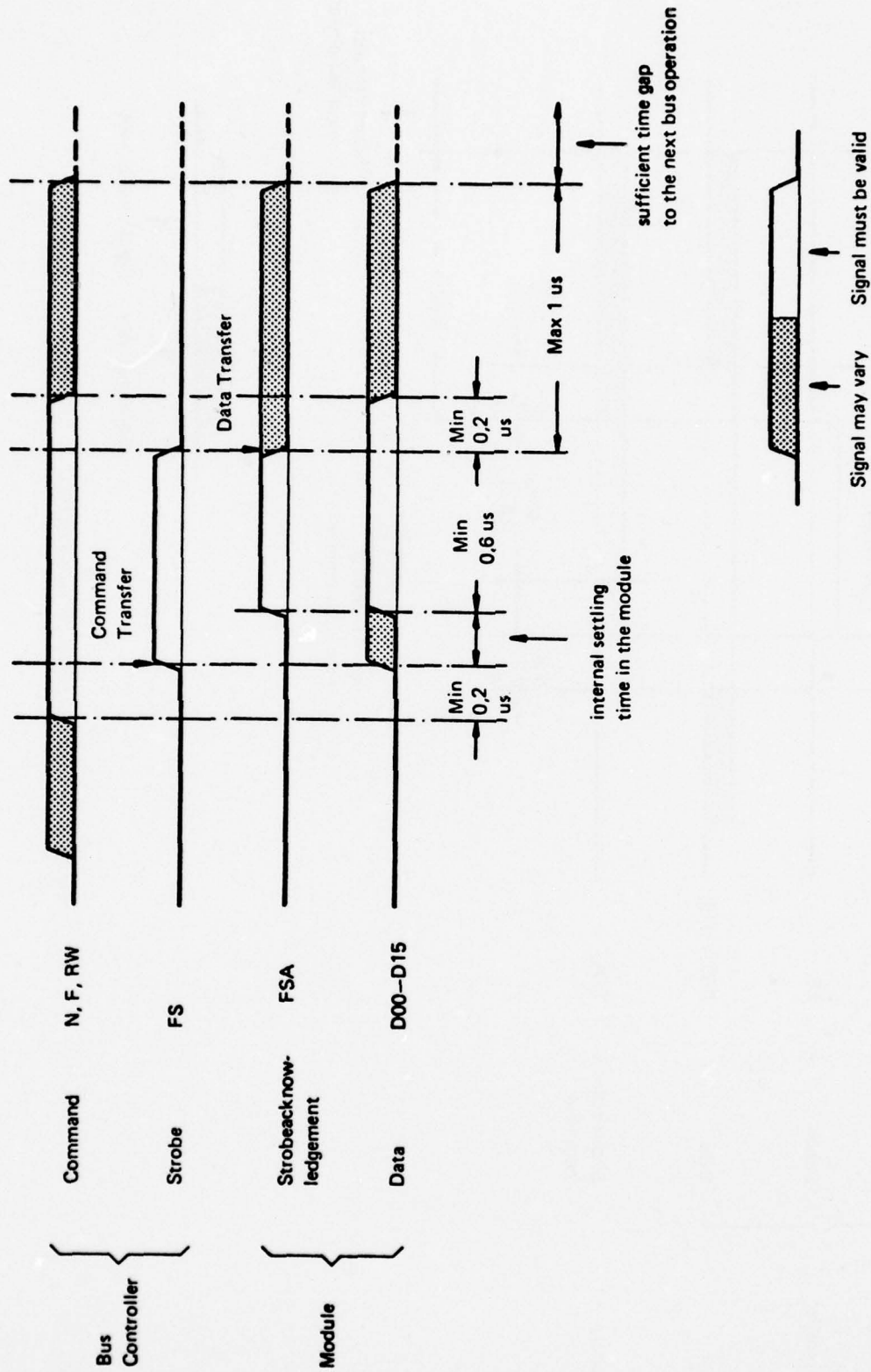


Fig.3 Read transfer operation on the parallel bus

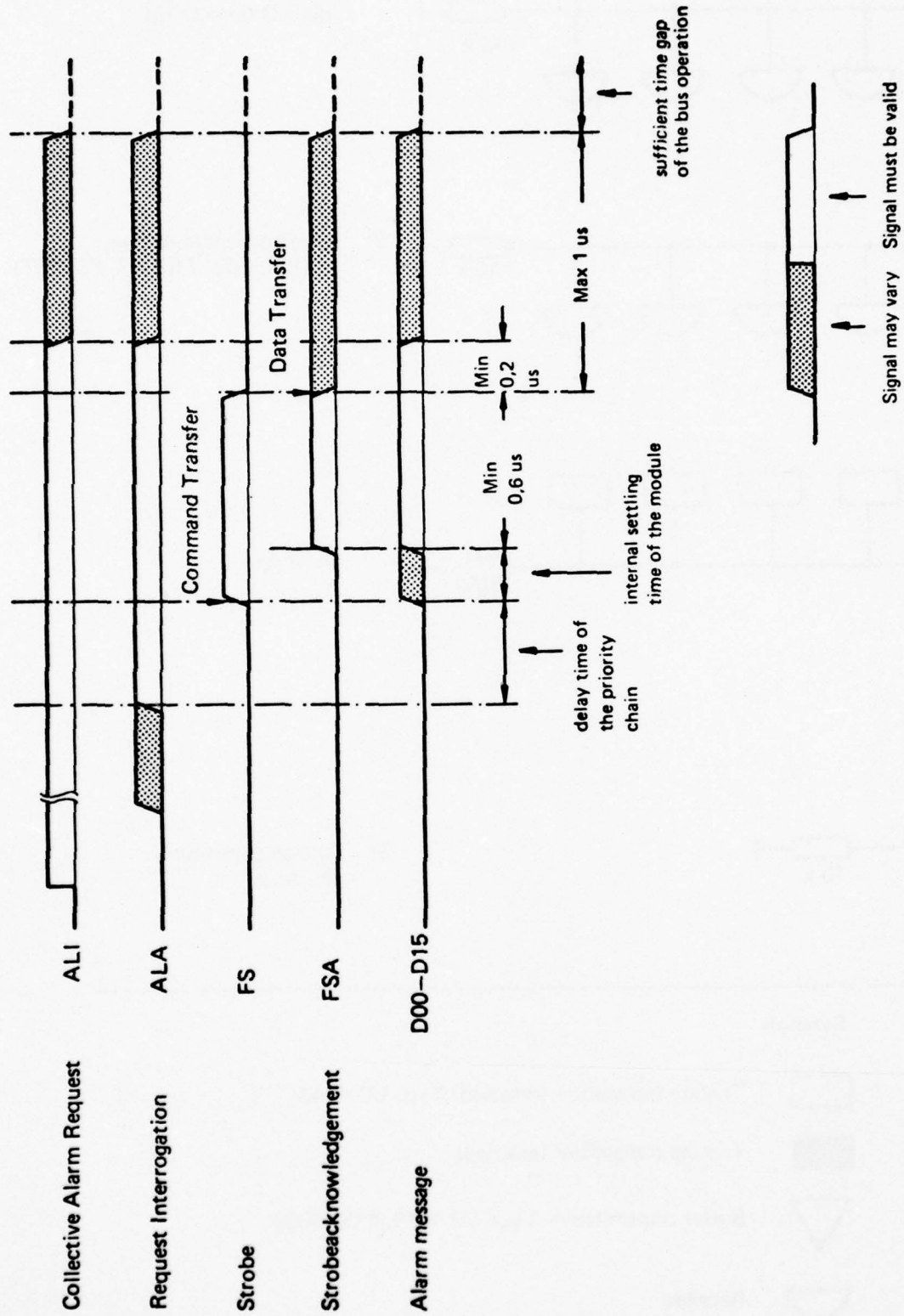
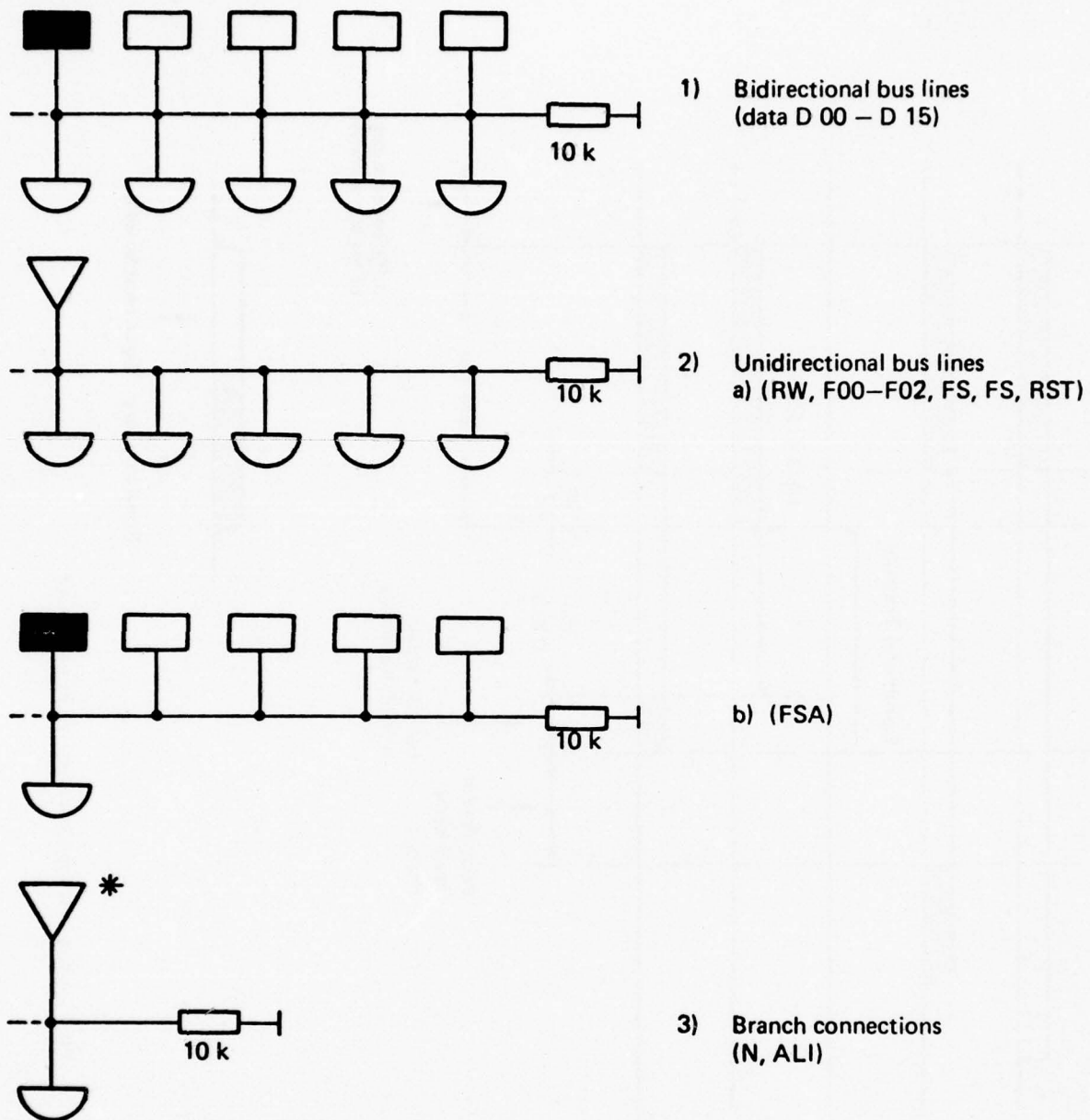


Fig.4 Alarm (interrupt) operation on the parallel bus



Symbols

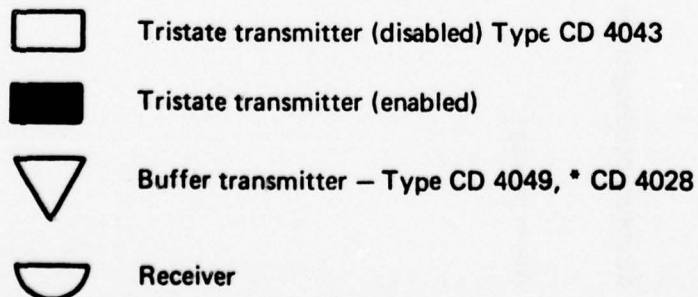
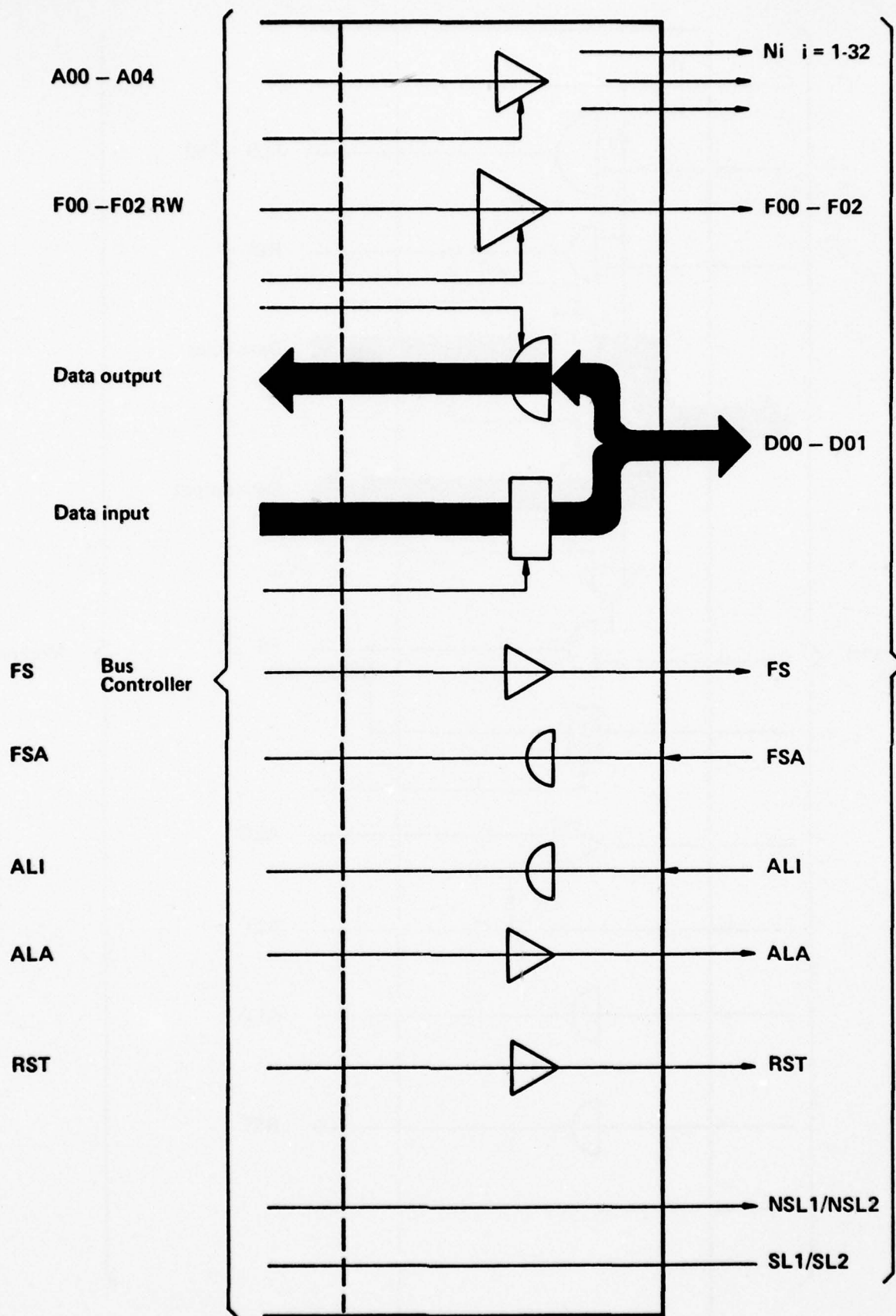


Fig.5 Data path transmitter/receiver configuration



Data path connections in data path controller (bus controller)

Fig.6 Interface to the control unit (bus controller)

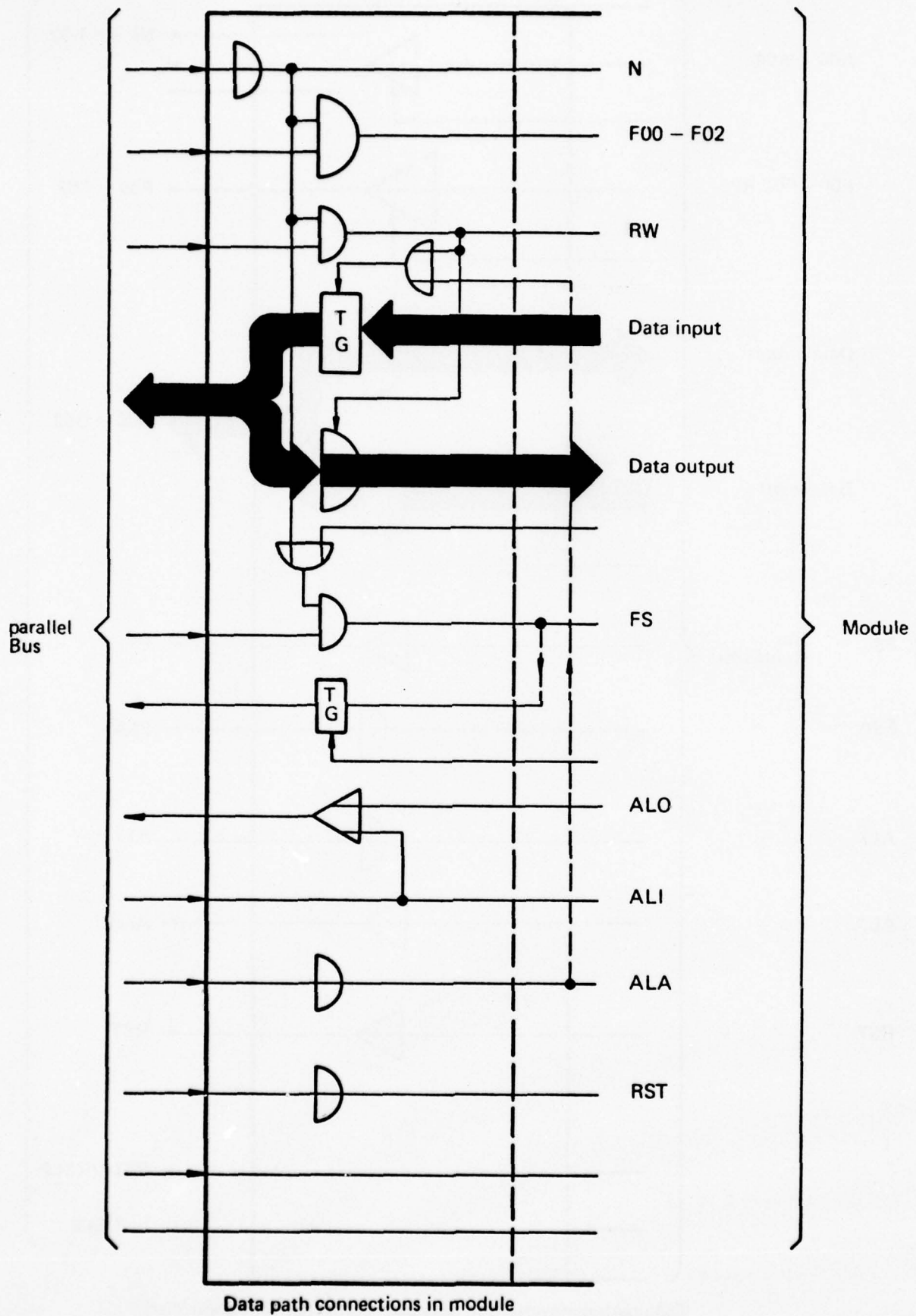


Fig.7 Intersections to the module

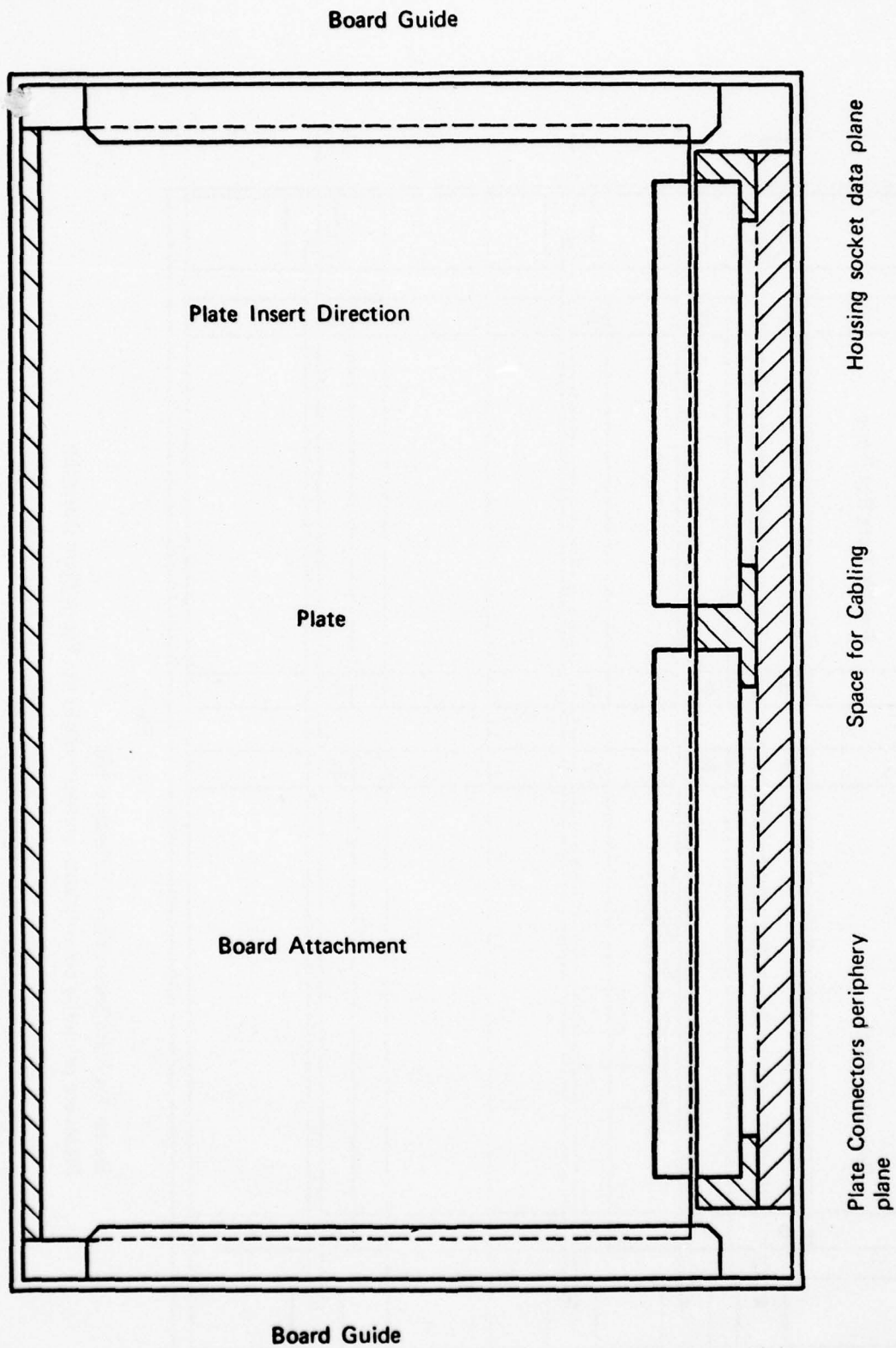
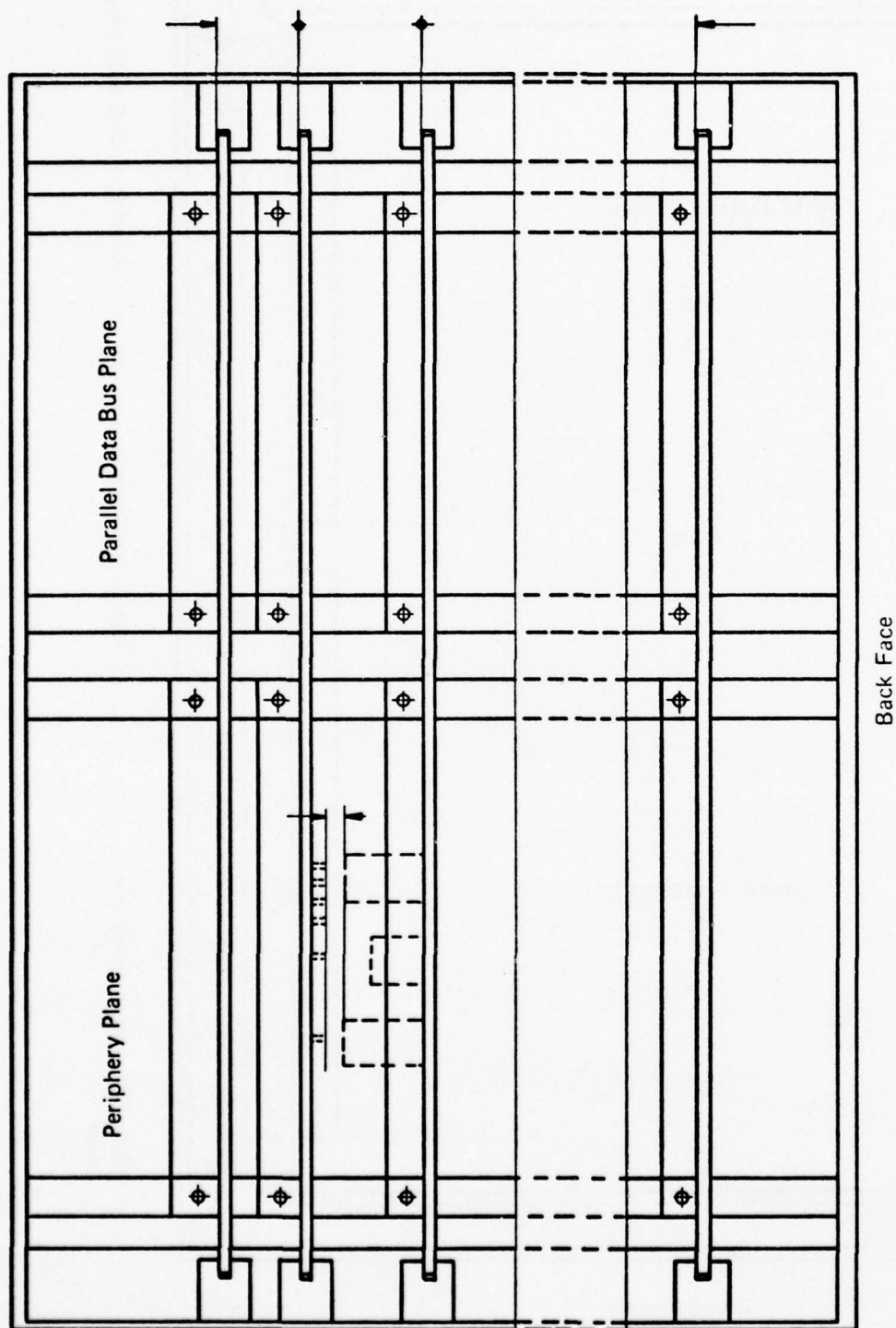


Fig.8 Sketch of board configuration



Back Face

Raster for Plate Separation "Preferably 1/4"

Plates are primarily covered with components in the Front Plate Direction

Fig.9 Dimensions of board configuration

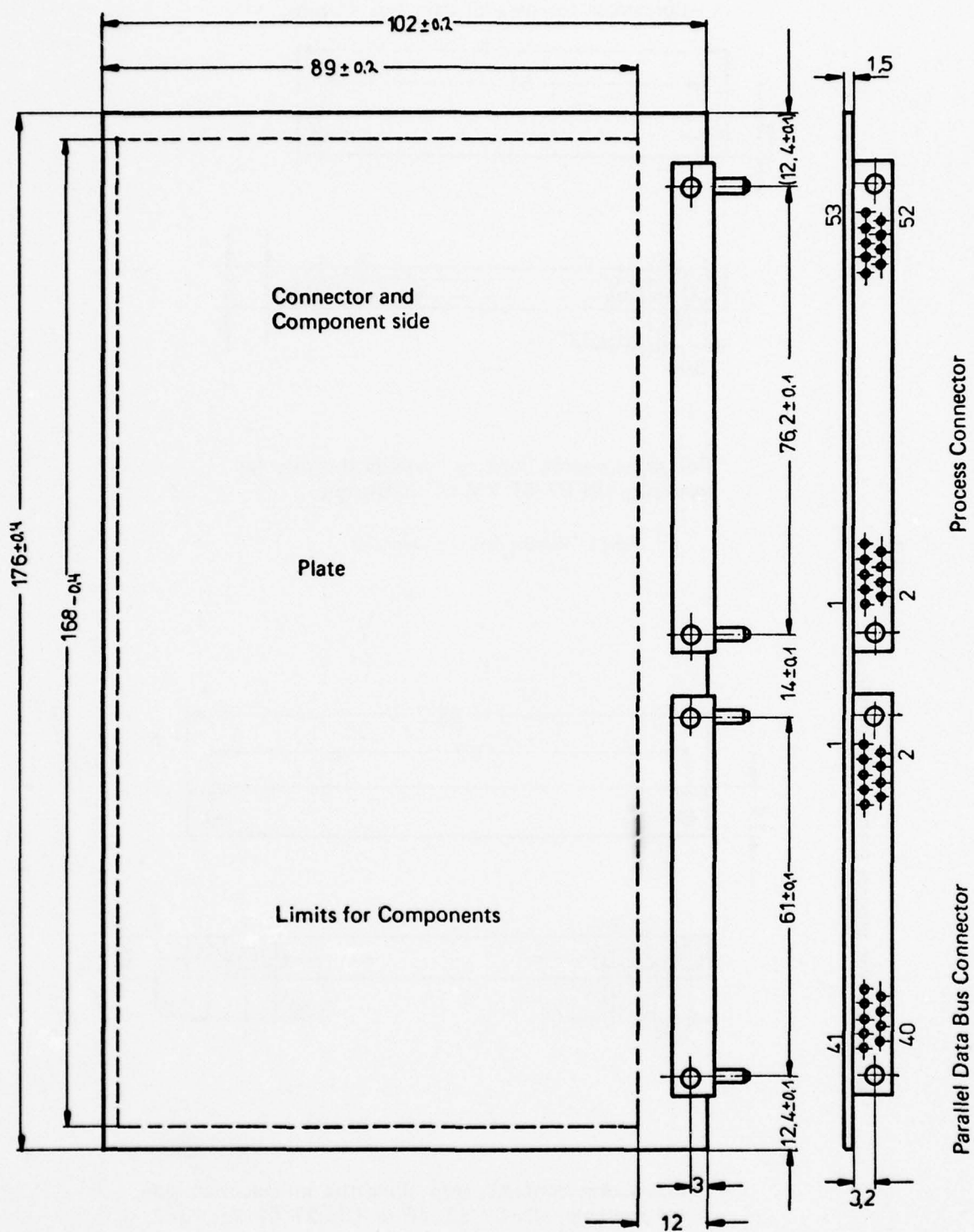
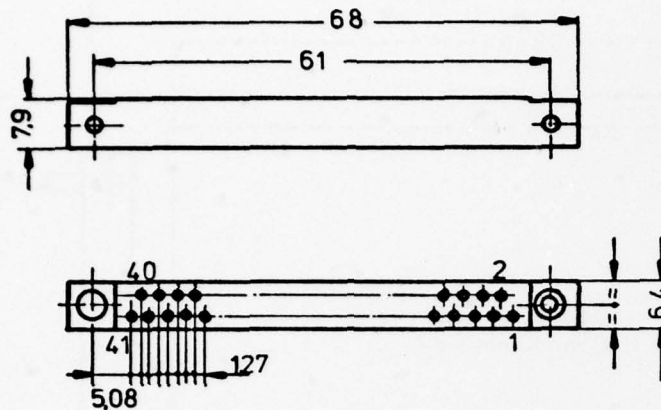


Fig.10 Board dimensions

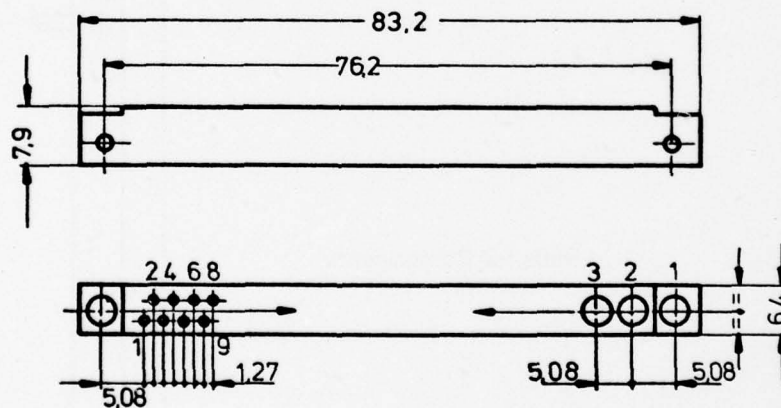
Connector for parallel data bus 41-pins;



For pin contacts, looking towards the pin, for example, UN 27 41 YM of Comtronic

Fig.11 Parallel data bus connector

Connector for process plane — 53 poles;



For socket contacts, seen along the introduction side, for example, UN 27 53 YF or UN 27 41 YF +3F3 (coax) made by Comtronic

Fig.12 Peripheral connector

ANNEX F

BUS ASSESSMENT AND COMPARISON

CONTENTS

	Page
1. INTRODUCTION	F-3
2. GENERAL OBSERVATIONS	F-3
PULLOUTS – A, B, C and D:	
INTRODUCTORY REMARKS	F-4/5
Ranges of Application	F-4/5
Special Features	F-4/5
BUS ARCHITECTURE	F-4/5
Logic Structure	F-4/5
Spatial Structure	F-4/5
Channelling	F-6/7
Number of Remote Terminals	F-6/7
TRANSMISSION METHODS	F-6/7
Transmission Gross Rate	F-6/7
Base Band Modulation	F-6/7
Bit Synchronization	F-6/7
Word or Message Synchronization	F-8/9
Error Detection	F-8/9
WORD AND MESSAGE FORMATTING	F-8/9
Addressing	F-8/9
Data Formats	F-8/9
BUS OPERATION AND BUS CONTROL	F-8/9
Mode of Transmission	F-8/9
Message Routing	F-10/11
Alarm Processing (Interrupt)	F-10/11
Coupling of the Stations to the Bus	F-12/13
System Failures	F-12/13

BUS ASSESSMENT AND COMPARISON

1. INTRODUCTION

This Annex is an attempt at a comparison of four serial bus systems. The purpose of the comparison is to point out the various bus concepts and the characteristics resulting therefrom. The following systems or system proposals were taken into consideration:

- (A) MIL-STD-1553A, "Aircraft Internal Time Division Command/Response Multiplex Data Bus", dated 30 April 1975.
- (B) Einheitliche Grundstruktur und Schnittstellen (funktionell, elektrisch, mechanisch) in Flugführungssystemen mit digitaler Signalverarbeitung. Draft for German recommendations, Issue April 1975.
- (C) "Recommandations provisoires d'un système de transmissions multiplexées sur lignes bus à bord d'aéronefs". (Preliminary recommendations for a transmission system with multiplex bus lines on board aircraft.) Prepared by an official technical commission in France, Issue April 1974.
- (D) Definition du DIGIBUS pour l'échange des informations numériques dans un Avion de Combat (Avions Marcel Dassault Bréguet Aviation).

2. GENERAL OBSERVATIONS

System A: MIL-STD-1553A is the result of three years effort to come up with a workable standard. Derived from the earlier EMUX version and experience from the F-15, B-1 and numerous hardware development contracts, this military standard reflects the final version approved by both government and industry. The system design has a simple structure and serves for transmission of data between remote terminals under central control. The system is so designed as to allow multiple busses as well as multiple controllers, where control can be dynamically switched between controllers both for better data transfer and for degraded mode operation. The system has alarm processing built into the status word and also provides for function (mode) commands. It can be designed (via software defined protocol) to be sink-oriented or source-oriented. In other words, the user of data can simply use the bus to collect data when needed, or a sensor can distribute the data to all potential users as it is generated. The standard provides a standard digital interface to subsystems. The USA is committed to this standard because of the large number of applications on which it is being used; e.g.: DAIS, F-16, F-18, AMST.

System B: This is characterized by high flexibility and system reliability, which, in principle, is achieved by providing many remote terminals each with the capability of being the bus controller, i.e. all those remote terminals participating in the system have direct access to the bus and many control it. Only one remote terminal, however, is active as a bus controller at any one time. Any deficiency in the active controller is countered by an automatic switch-over to another controller. Intensive address and data protection results in high transmission reliability. The system concept is independent of the transmission medium used. At present coax cables or twisted pair cables are recommended; a later changeover to optical wave guides, for example, is uncomplicated. This system is a preliminary recommendation, and no final decision has so far been made by the German MOD authorities.

System C: "Recommandations provisoires d'un système de transmissions multiplexées sur lignes bus à bord d'aéronefs" is not a firm specification for a specific bus system; it is a set of recommendations for possible design variants with evaluations. It was included in the bus assessment and comparison as it nevertheless offers important suggestions for consideration.

System D: This is one realization within the frame of the recommendations given in System C. The system is characterized by inherent high reliability which is achieved by the specified redundancy. High flexibility is built in by the use of sophisticated command and data codes.

A
MIL-STD-1553A

INTRODUCTORY REMARKS

Ranges of Application

Originally intended for military avionics only, to simplify integration, installation, test and retrofit, it has now broadened to other aircraft subsystems such as flight control, electrical power management, stores management, built-in-test and commercial entertainment systems. In addition, it is being used on ships and ground vehicles and is now being considered for ground inter-shelter communications. Initially, it was sold on its weight and installation cost savings; however, its flexibility is now the prime selling point.

Special Features

The system is capable of being configured with one or more busses, redundant and/or hierarchical, with slave or master terminals in any combination. Any master can control the bus; slaves can only respond. The dynamic bus allocation feature can be used where different masters need to control the same bus. Alarm processing (interrupt) is now an inherent feature as is the use of functional commands (i.e. instructions from the controlling master). Another special feature of great importance is the standard digital interface to the subsystem. A serial digital bus is defined and so is the discrete signal interface. A parallel bus is built using discretes and their control signals in parallel.

BUS ARCHITECTURE

Logic Structure

The system is built up hierarchically. The "master terminal" controls the bus at any instant of time and all other terminals, alternate masters included, are slaves. Control of the bus can be dynamically handed to alternate masters, but there can be only one master at any one time. When in charge, the master initiates and monitors all bus traffic. When redundant busses are used, a different master can administer over each separate bus. When at any instance a terminal is master over one bus, it will simultaneously be a slave to all others. If a terminal receives two or more commands simultaneously, an order of priority of response can be set up. A designated master or a separate control electronics box may be set up to assign bus masters and manage priorities in case of bus or terminal failure (Degraded Mode Management).

Spatial Structure

The bus is linear in its maintrunk, maximum length of 100 metres with up to 32 tees to connect remote terminals each having a length of between 0.35 and 6.65 metres.

B
**Einheitliche Grundstruktur und Schnittstellen –
Funktionell, Elektrisch, Mechanisch – in
Flugführungssystemen mit Digitaler Signalverarbeitung**

Ranges of Application

The serial bus system is intended to be used for interconnection of the remote terminals of guidance and control systems, to replace the wire bundles and to ease retrofit programs. Its purpose is mainly to interconnect subsystems; therefore, the system architecture is decentralized and not hierarchical.

Special Features

Decentralized bus administration, alarm processing. The active bus controller administers the bus, triggers bus operations, and monitors them. The system comprises mechanisms for automatic or controlled transfer of bus control from the active master to another remote terminal with the capability of being a bus controller.

Logic Structure

The system distinguishes between remote terminals that can only receive information and those that can receive and transmit by interrupt. There are remote terminals with the capability of being a bus controller. Only one master station can be active as controller at any one time.

Spatial Structure

Bus with linear extension, maximum length of 200 m. Stubbing not specified.

**Recommendations
Transmission**

Ranges of Application

Data transmission terminals in military systems designed to replace wire bundles especially to save weight.

Special Features

Not discussed

Logic Structure

The system distinguishes between remote terminals that can only receive information and those that can receive and transmit by interrupt. There are remote terminals with the capability of being a bus controller. Only one master station can be active as controller at any one time.

Spatial Structure

Several variants

nd Schnittstellen –
mechanisch – in
der Signalverarbeitung

C
**Recommendations Provisoires d'un Systeme de
Transmission Multiplexées sur Lignes Bus à
Bord d'Aéronefs**

D
**Definition du DIGIBUS pour l'Echange des
Informations Numeriques dans un avion
de Combat**

ended to be used
terminals of
replace the wire
ams. Its purpose
ems; therefore,
alized and not

Ranges of Application

Data transmission path to interconnect remote terminals in military aircraft. The system is designed to replace the wiring hitherto used, especially to save weight and installation cost.

Special Features

Not discussed.

Logic Structure

The system is built up strictly hierarchically. A "master station" controls the bus, and all other remote terminals are slaves. All bus operations are initiated and monitored by the master station as bus controller. This hierarchy is fixed and cannot inherently be changed by the system.

Spatial Structure

Several variants.

tion, alarm
oller administers
and monitors them.
s for automatic or
from the active
al with the capability

tween remote
information and those
interrupt. There are
ility of being a bus
ion can be active as

maximum length
d.

4)

Ranges of Application

Data transmission on board combat aircraft to interconnect avionic equipment. The multiplex system is to replace the wire bundles used hitherto, to save weight and integration cost. Redundancy in the bus and bus controller is provided for, which results in the high reliability and integrity required for onboard use.

Special Features

Features for redundant systems are provided. Normal system operation comprises two independent bus systems, one for normal operation and one for emergency. Means are being provided for queueing of remote terminals wishing to be serviced at the same time.

Logic Structure

The system is built up hierarchically. In normal operation the main bus controller controls the bus. In case of failure an emergency bus controller takes over. Data exchange takes place normally by direct module to module transfer.

Spatial Structure

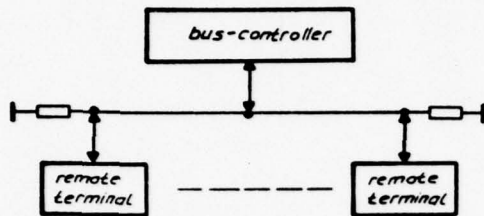
Bus with linear extension redundancy of the bus lines are provided. Short tees (less than 30 cm) are prescribed. Tees up to 50 cm are an exception.

2)

A

Channelling

Bidirectional two wire circuit of twisted and screened (shielded) wire, 90pF/m, $70 \pm 10\%$ Ohms characteristic impedance terminated at both ends with Z_0 .

**Number of Remote Terminals**

Thirty-two (32) terminals are maximum. This is the summation of masters and slaves. One master with 31 slaves is one extreme; 32 masters is the other. Three terminals (with growth in mind) are a minimum to justify the use of the data bus concept. Minimum of one bus master required. If more than one master is connected to the bus, others will be in the slave mode because there can be only one master at any one time. Masters can generate commands and respond to commands from other sources; slaves can only respond to commands received.

TRANSMISSION METHODS**Transmission Gross Rate**

1 Mbit/sec

Data flow is asynchronous. When transmitting, the data rate is 1.0 Mbit/sec; otherwise, the bus is idle. Experience shows a duty cycle of 25–50%.

Base Band Modulation

Manchester Bi-phase code

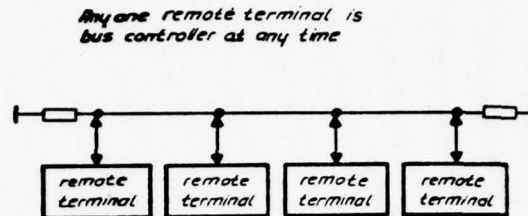
Bit Synchronization

Extracted from the self clocking Manchester bi-phase code.

B

Channelling

Bidirectional two wire circuit. Transmission media not yet specified. Variants proposed are coax or twisted shielded wire cables terminated at both ends with Z_0 , $60 \pm 10\%$ Ohms.

**Number of Remote Terminals**

Max. 64 remote terminals.

Transmission Gross Rate

1 Mbit/sec

Data flow is asynchronous. When transmitting, the data rate is 1.0 Mbit/sec; otherwise, the bus is idle with zero voltage level.

Base Band Modulation

Manchester Bi-phase code with additional zero voltage level.

Bit Synchronization

Extracted from the self clocking Manchester bi-phase code.

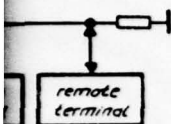
41

C

Channelling

Several variants.

Transmission
proposed are coax
terminated at both



Several variants.

Number of Remote Terminals

Not specified.

Transmission Gross Rate

1 Mbit/sec

When transmitting,
wise, the bus is

Base Band Modulation

Manchester Bi-phase code, Bi-phase RZ code,
Litton code are proposed as candidates.

Bit Synchronization

Extracted from the self clocking bit codes,
several proposals: Manchester bi-phase; Bi-phase-
RZ; Boeing-Litton.

h additional zero

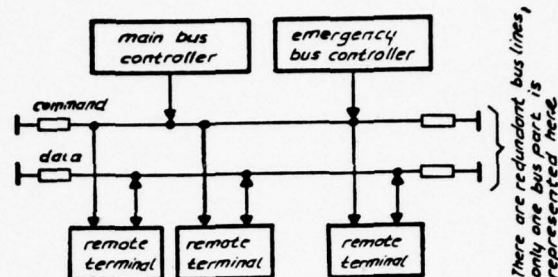
ing Manchester

4

D

Channelling

Uni-directional wire circuit of twisted pair for
transmitting commands from the bus controller to
the terminals, bidirectional wire circuit of twisted
pair for transmission and receiving data. The
characteristic impedance of the transmission line
is 75 Ohms. The cables are terminated at both
ends with Z_0 .

**Number of Remote Terminals**

Max. 32 remote terminals.

Transmission Gross Rate

1 Mbit/sec

Data flow is asynchronous. When transmitting,
the data rate is 1.0 Mbit/sec.

Base Band Modulation

Bi-phase resetting code with three level
modulation.

Bit Synchronization

Extracted from the self clocking bit code on
the procedure line.

A

Word or Message Synchronization

Each word is marked by a special sync character. An invalid bit representation of the data code is used.

Error Detection

1 parity bit per address and data word (16 bits). Additional error detection by redundant bit code (Manchester II) and sync character before each word. Reply for the transmission is given by the receiver. The reply contains the receiver address and the error status.

WORD AND MESSAGE FORMATTING

Addressing

The address information is contained in the command part. Address of the selected station: 5 bits, i.e. a maximum of 32 stations. Subaddress 5 bits, 1 bit transfer direction. Five bits are used for the word count. (Number of sequential words to be read in/out of the remote terminal.) Maximum addressing capability at any one remote terminal is 31 blocks, each block having up to 32 words (1024 words). Also, can transmit up to 31 mode commands (instructions to the terminal) using the reserved all-zero subaddress. (If the subaddress is 00000, then the 5 bit word count field is decoded as an instruction. The all-zero word count field in conjunction with an all-zero subaddress signifies a data bus master control transfer offer.) A distinction is made between data operations and command operations by means of special sync characters which comprise three bit times.

Data Formats

Data, address, command, status messages are word-oriented. Each word comprises a sync character, 16 data bits, and parity bit. Commands and status messages use the same sync character, data words the inverted character. Several data words can be combined in one data block. The length of this block is given by the pertinent command.

BUS OPERATION AND BUS CONTROL

Mode of Transmission

Asynchronous, half duplex, only one remote terminal or the bus controller transmitting on the bus at any one time.

B

Word or Message Synchronization

For synchronization, a third state is used, which can be distinguished from the data state 0 and 1, e.g. the "pause" state or no information. The zero level voltage is recommended (synchronization by envelope detection).

Error Detection

Error detection cyclic code (8 bits) per part message, i.e. separate for the address portion and the data portion. Reply for the transmission is given by the receiver. The reply includes the receiver address and an error status. The replies are protected as well. Additional protection by redundant bit code (Manchester II).

Addressing

Address of the selected station: 6 bits, i.e. a maximum of 64 stations, 5-bit subaddress or function code, a further 5 bits for group addressing, direction identification, special functions.

Data Formats

The traffic on the bus is effected word by word. The length of the word depends on the function; it is, however, always a multiple of one byte (8 bit sign). The shortest word is 2 bytes long. Greater lengths are permitted; they should, however, be agreed upon implicitly in the system between the participating remote terminals.

Mode of Transmission

Asynchronous, half duplex, only one remote terminal on the bus transmitting at any one time.

(1)

B

Synchronization

...n, a third state is used,
...ed from the data state 0
...state or no information.
...recommended (synchroniza-
...on).

...lic code (8 bits) per part
...r the address portion and
...y for the transmission is
...The reply includes the
...error status. The replies
...Additional protection by
...chester II).

...ted station: 6 bits, i.e. a
...5-bit subaddress or function
...r group addressing, direction
...ctions.

...us is effected word by word.
...depends on the function; it
...multiple of one byte (8 bit
...is 2 bytes long. Greater
...y should, however, be
...the system between the
...inals.

...duplex, only one remote
...mitting at any one time.

C

Word or Message Synchronization

Several proposals: synchronization via invalid wave forms, sync character as special bit combination, special sync signal level.

Error Detection

Proposal: parity bit; not explained in detail.

Addressing

Not fixed.

Data Formats

The word length is a multiple of one byte (8 bit sign).

Mode of Transmission

Half duplex, asynchronous.

D

Word or Message Synchronization

Pauses with a minimum of one character between words. Synchronization by envelope detection, at word level by counting and format checks.

Error Detection

Parity bit for each character. Additionally redundant bit codes. One character contains 10 bits gross information.

Addressing

Via the command bus. The bus controller selects the remote terminals with an address code.

Five bits are used for addressing, i.e. a maximum of 32 remote terminals. Besides these five address bits, additional three bits for special functions, and eight bits with additional information are contained in a command word.

Data Formats

Command and data information are word-oriented. A command word contains 2 characters, a data word contains four characters. Block transfer is possible. Each character contains eight information bits, 1 bit for identification of the type of character and 1 parity bit.

Mode of Transmission

Half duplex, asynchronous.

(1)

Message Routing

Each bus operation is initiated and monitored by the bus controller.

The remote terminals are not autonomous. No simultaneous traffic is envisaged between the bus controller and remote terminals at the same time.

All remote terminals addressed respond with a status word.

Transfer: bus controller — remote term.

The bus controller transmits the "receive" command to the remote terminal, followed by a series of data words. The "receive" command contains the remote terminal address, the sub-address, and the number of ensuing data words.

A status word as response is added to the received data by the addressed remote terminal. The status word is composed of the station address and an error code.

Transfer: remote term. — bus controller.

The bus controller transmits a request to the remote terminal to transmit. The remote terminal replies with the status word followed by data words.

Transfer: remote term. — remote term.

The bus controller transmits a "receive" request to the receiving remote terminal and a transmit request to the transmitting remote terminal. Afterwards the sending remote terminal transmits the data.

Alarm Processing (Interrupt)

Since all transmissions include status words, the terminals will take one of the status word "to be assigned" bits and set it to "1" when interrupt processing is required by the terminal. Since the status word bits are software-interpreted by the master, priorities are programmed into its software. During normal data transmissions, not all terminals are accessed frequently enough to see the interrupt (alarm) bit. Therefore, at least once per second all terminals are addressed to send their status words for built-in-test (BIT) purposes. If this alarm time takes too long, it can be programmed to be sampled at a higher rate. Masters can interrupt, also, by using the functional mode (instruction)/command. Critical interrupts will be hardwired.

Message Routing

Several remote terminals can be bus controllers. Only one bus controller is active at a given time. Bus operations are initiated and controlled by the active bus controller. Bus control can be autonomously passed over from the bus controller to another station via an automatic priority chain.

Transfer: bus controller — remote term.

The bus controller transmits the address, the function code and the data to the remote terminal; the remote terminal gives its response with its address.

Transfer: remote term. — bus controller.

The bus controller transmits the address and the function code to the remote terminal; the remote terminal replies with its address and the requested data.

Dynamic transfer of bus control.

The remote terminal with a capability to be a bus controller observes the bus. If the bus is not occupied the remote terminal acquires bus control according to its priority, and transmits its information. This takes place as a normal bus operation.

Transfer: bus controller — several remote term.

The same as the transfer from bus controller to one remote terminal except that a global command is given and no response is received.

Transfer: remote term. — remote term.

If a remote terminal to remote terminal transfer is required, bus control is handed over to the remote terminal which is to transmit. After this, a normal bus controller to remote terminal transfer takes place.

Alarm Processing (Interrupt)

Each remote terminal with interrupt capability can request service. After any interrupt request is located, its priority is determined and the remote terminal gets the requested service after the current message transfer is completed.

Message

Not

Transfer

Transfer

Transfer

Transfer

Alarm

N

C

D

Message Routing

Not fixed, several proposals.

Transfer: bus controller – remote terminal.

Transfer: bus controller – several remote terminals.

Transfer: remote terminal – bus controller.

Transfer: remote terminal – remote terminal;

Message Routing

Each bus operation is initiated and monitored by the bus controller. The remote terminals are not autonomous. No simultaneous traffic is envisaged between the bus controller and remote terminals at the same time.

All remote terminals explicitly addressed respond with a status word.

Transfer: bus controller – remote term.

The bus controller transmits the receive command(s) to the remote terminal(s), containing the address, an identification field and a function, followed by an additional instruction and possibly delay phase.

A status word as response is given to the bus controller after the preselection instruction phase via the data line. The data transfer on the data line is accompanied by service characters on the procedure line.

Transfer: remote terminal–bus controller and remote terminal–remote terminal is similar.

Alarm Processing

Not dealt with.

Alarm Processing

No provisions are foreseen for alarm processing of signals from the remote terminals.

A

Coupling of the Stations to the Bus

Isolated

Technical solution: transformer coupling. By dividing the stations into a logic part and a separate bus coupler, the system allows any technical solution for the coupling.

System Failures

Highly recommends dual redundant data busses with a minimum of two masters. If a bus is damaged or transmitter/receiver logic fails at any terminal, data access is through the redundant bus. Master failure will allow redundant master to take over. All data source or data processing failures will cause degraded operation; wire shorts/opens or MTU failures will cause bus switching to alternate busses retaining 100% transmission capability. System redundancy is a function of integration priorities, to be determined by the system designer.

B

Coupling of the Stations to the Bus

Isolated

Technical solution: transformer coupling. By dividing the stations into a logic part and a separate bus coupler, the system allows any technical solution for the coupling.

System Failures

Decentrally organized system; i.e. a failure of a master station does *not* result in a system failure. The system degenerates with *each* station failure.

Coupling

Isolated

Technical
recommend

As t
and a sep
solution

System F

Cent
the centr
failure of
stations l

Rec
central st
the redund

C

D

the Bus

transformer coupling.
 a logic part and a
 system allows any
 coupling.

system; i.e. a failure of
 result in a system failure.
 each station failure.

Coupling of the Stations to the Bus

Isolated

Technical solution: transformer coupling recommended.

As the stations are divided into a logic part and a separate bus coupler, any technical coupling solution can be applied to the system.

System Failures

Centrally organized system; i.e. a failure of the central station — the master — results in a failure of the entire system. Failures of the other stations lead to system degeneration.

Recommended redundant bus. If a bus or a central station is damaged, data access is through the redundant unit.

Coupling of the Stations to the Bus

Isolated

Technical solution: transformer coupling. As the bus coupler provides the standard functions to the bus operation and special-to-type functions are performed by the subsystem coupler, the bus system allows any technical solution.

System Failures

Remote terminal and bus controller operation is monitored continuously. Standby units are also monitored. In case of failures, the system deactivates the failed unit and switches to the remaining unit (bus line or bus controller). System has high inherent software and hardware built-in integrity.

ANNEX G**JOVIAL J73/1**

CONTENTS

	Page
<u>CHAPTER 1. INTRODUCTION</u>	G-5
PURPOSE OF THE MANUAL	G-5
OVERVIEW	G-5
THE DESCRIPTIVE METALANGUAGE	G-5
 <u>CHAPTER 2. BASIC ELEMENTS</u>	 G-7
JOVIAL SOURCE	G-7
SYMBOLS	G-8
ABBREVIATIONS	G-8
KEY WORDS	G-8
SIGNS, OPERATORS	G-9
Arithmetic Operators	G-10
Relational Operators	G-10
Logical Operators	G-10
Brackets	G-10
Address Operator	G-10
Assignment Operator	G-10
Separators, Terminators	G-10
NAMES	G-10
VALUES	G-11
NUMBERS, CONSTANTS	G-11
Numeric Constants	G-12
Bit Constants	G-12
Character Constants	G-13
Status Constants	G-13
BLANK, COMMENTS	G-15
 <u>CHAPTER 3. DATA DECLARATIONS</u>	 G-16
DATA ALLOCATION	G-16
ITEM DESCRIPTIONS	G-16
ITEM DECLARATIONS	G-17
TABLE DECLARATIONS	G-18
Table Dimensions	G-18
Table Structure	G-18
Table Packing	G-19
Constant List	G-20
Ordinary Table Declaration	G-21
Specified Table Declaration	G-22
BLOCK DECLARATIONS	G-23
OVERLAY DECLARATION	G-25
 <u>CHAPTER 4. FORMULAS</u>	 G-27
VARIABLES	G-27
Based Data	G-27
Function Variables	G-28
Variable Types	G-29
FUNCTIONS	G-29
Intrinsic Functions	G-30
Bit Function	G-30
Byte Function	G-30
Loc Function	G-30
Abs Function	G-31
Sign Function	G-31
Shift Function	G-31
Size Function	G-31
Data Size Function	G-32

	G-3
	Page
FORMULAS	G-32
CHARACTER FORMULAS	G-32
BIT FORMULAS	G-32
Logical Formulas	G-32
Relational Formulas	G-33
Conditional Formulas	G-33
NUMERIC FORMULAS	G-33
EVALUATION ORDER, PRECEDENCE	G-35
CONVERSIONS	G-35
NUMBER FORMULAS	G-36
EXAMPLE FORMULAS	G-36
 <u>CHAPTER 5. STATEMENTS</u>	 G-37
ASSIGNMENT STATEMENT	G-37
GOTO STATEMENT	G-38
STOP STATEMENT	G-38
RETURN STATEMENT	G-38
IF STATEMENT	G-39
SWITCH STATEMENT	G-39
LOOP STATEMENTS	G-40
While Loop	G-41
For Loop	G-41
PROCEDURE CALL STATEMENT	G-43
 <u>CHAPTER 6. PROGRAM, PROCEDURE DECLARATIONS</u>	 G-45
NAME SCOPE	G-45
PROCEDURE DECLARATION	G-45
Procedure Data	G-46
Based Procedures	G-46
Procedure Parameters, Function Results	G-47
Procedure Body	G-48
DEFINE DECLARATION	G-49
NAME DECLARATION	G-51
EXTERNAL DECLARATIONS	G-51
 <u>CHAPTER 7. COMPOOL</u>	 G-53
COMPOOL DIRECTIVE	G-53
COMPOOL SOURCE	G-54
COMPOOL OUTPUT	G-55
 <u>CHAPTER 8. DIRECTIVES</u>	 G-56
SOURCE DIRECTIVES	G-56
Copy Directive	G-56
Skip, Begin, End Directives	G-56
LISTING DIRECTIVES	G-57
LINKAGE DIRECTIVE	G-57
TRACE DIRECTIVE	G-58
 <u>APPENDIX A. SUMMARY SYNTAX</u>	 G-59
<u>APPENDIX B. COMPILER ERROR MESSAGES</u>	G-105
<u>APPENDIX C. COMPILATION AND EXECUTION OF J73/I PROGRAMS</u>	G-111
<u>APPENDIX D. J73/I DEC-10 RUN TIME CONVENTIONS</u>	G-115

	Page
<u>APPENDIX E. J73/I HBC RUN TIME CONVENTIONS</u>	G-117
<u>APPENDIX F. J73/I COMPILER LIMITS, CAPACITIES AND RESTRICTIONS</u>	G-118
<u>APPENDIX G. J73/I TARGET MACHINE PARAMETERS</u>	G-119
<u>APPENDIX H. ASCII CHARACTER SET</u>	G-120

CHAPTER 1. INTRODUCTION

PURPOSE OF THE MANUAL

This manual specifies the level I subset of the JOVIAL 73 language, abbreviated as J73/I or J73, and serves as a guide to use of the language and its compilers. This description of J73 is intended primarily as a reference manual and not as a rigorous formal language specification. The manual is not structured as a tutorial; however, a considerable use is made of practical examples to assist in describing the language.

OVERVIEW

JOVIAL is a language which has evolved considerably from its inception in 1959. Designed originally for the purpose of command and control system programming, its primary area of applicability is for the development and maintenance of large systems. JOVIAL is especially suited to systems requiring efficient processing of a large volume of data of complex structure.

This manual comprises eight chapters which provide a machine independent description of J73 and various appendices containing a summary cross-referenced syntax, system dependent parameters, idiosyncrasies, compiler limits, and a guide to compiler usage.

THE DESCRIPTIVE METALANGUAGE

The J73 language is described in this manual using the following metalanguage:

- J73 source characters are represented in their standard forms; letters are upper case and β is used to denote the blank or space character.
- Metalinguistic elements are represented as one or more underscored lower-case names.
- The syntax of the language is specified by metalinguistic statements which in succession defines each metalinguistic element. The statements are intended to provide a clear, concise description of the permissible J73 forms with a minimum of notation.
- English language paragraphs and descriptive notes are used to further support the syntax description and to state the semantics associated with the element being described. Within this prose description the metalinguistic elements are not underscored.
- The metalinguistic notation used is outlined below:

$::=$ signifies that the element on the left is defined by the metalinguistic expression on the right. An expression consists of J73 characters, metalinguistic elements, and metalinguistic symbols indicating choice, repetition, and optionality. Spaces are permitted within a string of J73 characters only if shown in the expression.

$\left\{ \begin{array}{l} \text{ } \\ \text{ } \end{array} \right\}$ denotes a choice of one of the vertically enclosed expressions.

μ when included as a choice in braces denotes that the expression may be omitted.

\sim indicates that the immediately preceding expression may be replicated with no separation between successive occurrences.

γ signifies replication of an expression is permitted with successive appearances separated by the character indicated by γ . γ will be a comma, a colon, or β for blank separation. If blanks are indicated, multiple blanks are permitted; omission of the blank separator is optional in many instances (described further in Chapter 2).

∇ at the end of a line (column) denotes continuation to the next line (or column). At the beginning of a line (column) denotes continuation from a prior line (column).

The examples below will illustrate the metalanguage and clarify its use.

$$\text{bit value} ::= \left\{ \begin{array}{l} 0 \\ 1 \end{array} \right\}$$

defines the element bit value to be the character 0 or the character 1.

$$\text{bit list} ::= \left\{ \frac{\text{bit value}}{\mu} \right\} \gamma$$

defines the element bit list to be a string of bit values (or nulls) separated by commas.

The following are sample lists:

0	legal
0,1	legal
0,0,1,,1	legal
,,,	legal
00,1	illegal
0,A,1	illegal

CHAPTER 2. BASIC ELEMENTS

JOVIAL SOURCE

A JOVIAL compilation is made up of declarations, directives, and statements. Declarations establish the data base description and define attributes of names. Directives provide compiler control and optimization information. Statements present the processing algorithms. Declarations, directives and statements are composed of symbols. Symbols are the words of the J73 language and comprise the following language elements:

letter ::= {
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
}

digit ::= {
0
1
2
3
4
5
6
7
8
9
}

mark ::= {
+
-
*
/
\
>
<
=
@
.
:
;
(
)
[
]
,
"
!
\$
}

$$\text{character} ::= \left\{ \begin{array}{l} \text{letter} \\ \text{digit} \\ \text{mark} \\ \text{other character} \end{array} \right\}$$

Note that letters are defined to be the upper case letters only. Marks are used in J73 either alone or in conjunction with other characters as operators, delimiters, and separators. Other characters are the remaining implementation dependent characters available on a system which are accepted within character constants and comments but otherwise have no intrinsic language definition. These characters are defined in Appendix A for each compiler.

Program source is considered as a continuous stream of characters; source line boundaries, if they exist, are ignored by the compiler.

SYMBOLS

Symbols are the words and punctuation marks of J73. Each symbol is made up of one or more characters. A blank is usually required to separate symbols; however, in certain instances, the blank may be omitted. Additional blanks may always be used where one is required as a symbol separator.

$$\text{symbol} ::= \left\{ \begin{array}{l} \text{abbreviation} \\ \text{sign} \\ \text{key word} \\ \text{name} \\ \text{number} \\ \text{constant} \\ \text{comment} \\ \text{directive key} \\ \text{status} \\ \text{define string} \end{array} \right\}$$

Except as noted in subsequent sections of this chapter, blanks are not permitted within symbols.

Certain character combinations have an intrinsic meaning in J73. These intrinsic forms are abbreviations, key words and signs. Other combinations take the meaning as described in a program. Examples of these are names, constants, comments.

ABBREVIATIONS

Certain letters are used as abbreviations to specify various attributes of program entities. These abbreviations will be described later in the context of their usage.

$$\text{abbreviation} ::= \text{letter}$$

KEY WORDS

Key words are used in J73 to provide the primary purpose or function of a statement or declaration. Key words are also used as operators, descriptors, and attributes.

Arithmetic Operators

+	Addition, unary plus
-	Subtraction, unary minus
*	Multiply
/	Divide
\	Modulo
**	Exponentiation

Relational Operators

=	Equal
<>	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Logical Operators

NOT	Logical complement
AND	Logical multiplication
OR	Logical addition, inclusive or
XOR	Logical subtraction, exclusive or
EQV	Equivalence

Brackets

()	Expression grouping list delimiters, status constants
[]	Position brackets, used for subscripts, switches, status lists, item declarations
" "	Comment, define brackets
' '	Constant brackets
BEGIN END	Compound statement, declaration delimiters

Address Operator

@	At address
---	------------

Assignment Operator

=	Assignment
---	------------

Separators, Terminators

:	Statement name terminator, overlay, dimension and parameter separator
,	List separator
;	Statement, declaration, directive terminator
!	Directive indicator

The usage and description of these signs and operators are discussed in Chapter 4.

NAMES

Names are used in J73 to apply to data elements, statements, and other programmer declared entities.

$$\text{name} ::= \left\{ \begin{array}{c} \text{letter} \\ \$ \end{array} \right\} \left\{ \begin{array}{c} \text{letter} \\ \text{digit} \\ \$ \end{array} \right\} \sim$$

A J73 name must be two or more characters in length; however, name uniqueness is determined by the first thirty-one characters. Remaining characters are ignored. Names may not duplicate a key word. Dollar signs in external names may have a special system dependent meaning (see external declarations). Some examples of names are shown below.

Legal names	AB	LONG'LONG'NAME	\$SYSTEM	TABL'1
Illegal names	A	1AB	TWO NAMES	ONE-WAY TABLE

VALUES

J73 does not dictate the representation of data values but does assume that all data will be represented as a series of binary digits (bits), grouped in characters as bytes, or otherwise mapped onto computer words. Allocation is performed in units of words; maximum data size is typically in units of full words. However, addressing is expressed in system dependent address units. On many machines this corresponds to words; on others, addressing is in terms of bytes or half words.

J73 provides for five types of data — character strings, bit strings, logical or unsigned integer values, signed integers, and floating point values.

Character strings are represented as sequences of bytes, each byte consisting of a series of bits. The number of bits in a byte is system dependent as is the internal bit encoding for the characters. A character value which fits within a word is always allocated to a word by the compiler; however, a programmer may override this rule in a specified table. Similarly, character values must be allocated on a byte boundary, except in a specified table for character values less than a word and allocated entirely to a word. Unpacked character values will always begin in the leftmost byte of a word. If a word is not filled, unused bits are undefined.

Bit values may arise from table variables, the bit function call, bit constants, or converted character, floating, or integer values. Their maximum size is system dependent. Use of bit values whose size exceeds the maximum size for logical (unsigned) values is restricted to the context of assignment, relational operators, or as the object of a bit function call. Bit values may, of course, be truncated to integer size for use in other operations.

Integer (logical) values are also represented as a string of bits. The size of integer values is the number of bits required to represent its value in binary. An extra bit is required for signed integers. The maximum size of integers (exclusive of sign) which may enter into arithmetic operations is system dependent and is one bit less than the maximum for unsigned integers and logical values.

Floating values are represented as two signed components; the first represents the precision bits of the value (referred to as the significand) and the second is the exrad (or exponent of the radix). The radix of a floating value is system dependent. Unless otherwise indicated, floating values will be maintained in the implementation defined standard length, usually single precision.

Except for character values, data is normally represented right-justified in its allocated space; this allocated space for unpacked table data and scalar values is the least number of words required to contain it. Table data may be packed densely such that no extra bits are allocated to the value or it may be medium packed. Medium packing is used as an effective compromise to compact data space requirements without sacrificing the data access inefficiencies attendant to densely packed data. Medium packing is system dependent and will be described in an appendix.

NUMBERS, CONSTANTS

Numbers are used in J73 to specify size, storage position, counts, and constant values.

$$\text{number} ::= \left\{ \begin{array}{l} \text{digit} \sim \\ (\text{number formula}) \end{array} \right\}$$

In all contexts where a number may appear, a compile time number formula enclosed in parentheses is allowed. The definition of number formula will be described in Chapter 4.

Constants are forms of data whose value remains unchanged during program execution. J73 permits constant values to be specified as numeric, character, and bit string.

$$\text{constant} ::= \left\{ \begin{array}{l} \text{numeric constant} \\ \text{bit constant} \\ \text{character constant} \end{array} \right\}$$

$$\text{numeric constant} ::= \left\{ \begin{array}{l} \text{integer constant} \\ \text{floating constant} \end{array} \right\}$$

$$\text{integer constant} ::= \left\{ \begin{array}{l} \text{number} \\ \text{status constant} \\ \text{qualified status constant} \end{array} \right\}$$

$$\text{floating constant} ::= \left\{ \begin{array}{l} \text{digit} \sim E \left\{ \begin{array}{l} + \\ - \end{array} \right\} \frac{\text{pten}}{\mu} \\ \left\{ \begin{array}{l} \text{digit} \sim . \\ \left\{ \frac{\text{digit}}{\mu} \sim \right\} \cdot \text{digit} \sim \end{array} \right\} E \left\{ \begin{array}{l} + \\ - \end{array} \right\} \frac{\text{pten}}{\mu} \end{array} \right\}$$

$$\leftarrow \left\{ \begin{array}{l} M \left\{ \begin{array}{l} + \\ - \end{array} \right\} \frac{\text{precision}}{\mu} \end{array} \right\}$$

$\text{pten} ::= \text{digit} \sim$

$\text{precision} ::= \text{digit} \sim$

Numeric Constants

Numeric constants represent numeric values. An integer constant specifies a whole number. The size of an integer constant is the number of bits needed to represent the value. Some sample integer constants are shown below.

1 10 13 7654321089

A floating constant represents a rational number. Its value comprises an integer component (which precedes the decimal point or E if there is no decimal point) plus the fractional part (which follows the decimal point) times ten raised to a positive or negative power (indicated as pten). The optional magnitude specifier following the M serves to indicate the required precision in bits to accurately represent the value. For compilers which support multiple precision, the appropriate precision will be used which will guarantee the accuracy indicated. Blanks are not permitted in a floating constant.

Some example floating constants are

1E5 2.1718 .031416E+2 10E-6M35

Bit Constants

A bit constant represents a bit value. A bit constant comprises a string of beads whose bead size in bits is indicated in the specification of the constant; the total size of the bit constant is the bead size times the number of beads indicated.

$$\text{bit constant} ::= \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \right\} B' \text{ bead} \sim '$$

$$\text{bead} ::= \left\{ \begin{array}{l} \text{digit} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \\ I \\ J \\ K \\ L \\ M \\ N \\ O \\ P \\ Q \\ R \\ S \\ T \\ U \\ V \end{array} \right\}$$

The beads of a bit constant may be specified as one to five bits in size. The digit preceding the B indicates the bead size; only those beads whose value will fit in the bead size indicated are permitted. Digits represent themselves as a bead value, the letters A–V are used to represent the values 10–31 (see table below).

<i>bead</i>	<i>minimum bead size</i>	<i>binary value</i>	<i>bead</i>	<i>minimum bead size</i>	<i>binary value</i>
0	1	0	G	5	10000
1	1	1	H	5	10001
2	2	10	I	5	10010
3	2	11	J	5	10011
4	3	100	K	5	10100
5	3	101	L	5	10101
6	3	110	M	5	10110
7	3	111	N	5	10111
8	4	1000	O	5	11000
9	4	1001	P	5	11001
A	4	1010	Q	5	11010
B	4	1011	R	5	11011
C	4	1100	S	5	11100
D	4	1101	T	5	11101
E	4	1110	U	5	11110
F	4	1111	V	5	11111

Some examples of bit constants are

1B'000111110011' 3B'0763' 5B'AFAL'

The first two constants have the same value and a size of 12. The latter constant is 20 bits in size. The following would be illegal bit constants (the illegal characters are underlined):

3B'1328' 6B'78' 4B'Y672' 2B'12_30'

Character Constants

Character constants are used to denote strings of character values.

character constant ::= ' character ~ '

Character constants may contain any character (including blanks) that are representable on a system. A prime character is represented within a character constant by two consecutive primes. The size of a character constant in bytes is the number of characters represented within the containing primes (two consecutive primes represent one character). The encoding of characters is system dependent but will usually be the ASCII encoding. The byte size is also system dependent.

Some sample character constants are

'A' '{other characters}' '13"CHARACTERS'

Status Constants

Status constants also represent integer values. Status constants utilize symbolic names to represent values; a status list declaration is used to associate a symbolic status name with a particular integer value.

status constant ::= V(status)

status ::= { key word }
 name
 letter

$$\begin{aligned}
 \text{status list declaration} &::= \text{STATUS } \text{status list name} \triangleright \\
 &\triangleright \left\{ \left[\begin{array}{c} + \\ - \\ \mu \end{array} \right] \text{integer} \right\} \text{status constant} \sim \triangleright \\
 &\triangleright \left\{ \left\{ \left[\begin{array}{c} + \\ - \\ \mu \end{array} \right] \text{integer} \right\} \text{status constant} \sim \right\} \tilde{\beta} ;
 \end{aligned}$$

status list name ::= name

integer ::= $\left\{ \begin{array}{l} \text{number} \\ \text{bit constant} \\ \text{'character'} \\ \text{qualified status constant} \end{array} \right\}$

The status list declaration specifies the value of a status constant by associating values with order in the list. The optional integer in brackets indicates the value of the status constant immediately following in the list; subsequent constants are assigned the next integral value, negative values step through zero as -2, -1, 0, 1, 2, etc. This sequential assignment continues until the next bracketed integer occurs in the list. The integer may leave gaps but may not cause duplication of a status constant value or cause resetting to a lower value than the last prior constant in the list. If the initial bracketed integer is omitted, the list begins with the value zero.

A status may duplicate a key word, other names or it may be simply a letter. Status constants must be unique within a status list, but may duplicate constants in another list.

Status constants may be used interchangeably as integer constants. However, the contexts where they may be used in their unqualified form is restricted. These contexts are listed below:

- As a preset for a variable whose item description specifies the containing status list name.
- As a value to be assigned to an integer variable whose item description qualifies the constant as above.
- As an actual input parameter corresponding to a formal input variable which qualifies the status constant as above.
- In the context of

$\left\{ \begin{array}{l} \text{name variable} \\ \text{function call} \end{array} \right\} \text{relational operator } \text{status constant}$

The named variable or the function result must be declared with an item description which qualifies the status constant.

In all remaining contexts, a qualified status constant must be used.

qualified status constant ::= $V \left(\left\{ \begin{array}{l} \text{status list name} \\ \text{item name} \\ \text{table name} \\ \text{table item name} \\ \text{procedure name} \end{array} \right\} : \text{status} \right)$

The status list name directly qualifies the constant; the latter four must have been declared with an appropriate item description which associates them with the status list. Sample status lists and constants are illustrated below:

```

STATUS NETWORKS V(CBS), V(NBC), V(ABC), V(PBS);
STATUS LIST [-2] V(M'TWO), V(M'ONE), V(ZERO)
              [4] V(P'FOUR), V(P'FIVE);

V(X)
V(LIST:P'FOUR)
V(TABLE)

```


BLANKS, COMMENTS

Blanks or spaces may not occur within symbols except in character constants, status constants, define strings, and comments. In certain symbol contexts, one or more blanks must be used to separate the symbols (see chart below).

✓ indicates one or more blanks required as separation		Right symbol begins with					
		digit	letter	\$	'	.	"
Left symbol ends with	digit	✓	✓	✓	✓	✓	
	letter	✓	✓	✓	✓		
	\$	✓	✓	✓	✓		
	'	✓	✓	✓	✓		
	.	✓	✓				
	"						✓

J73 programs may be annotated with comments. Comments are legal between any pair of symbols, except in association with a define declaration or define parameter as noted in the sequel, and may be used as blanks for separation.

comment ::= " character ~ "

A comment may not contain a quote mark. Some example comments are shown below:

"THIS PROCEDURE USES A SERIES EXPANSION TO COMPUTE THE SINE OF ALPHA."

"ALL characters ARE LEGAL IN COMMENTS."

AD-A044 915

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 3/1
A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CON--ETC(U)
MAY 77 G E SCHWEIZER, A A CALLAWAY, E C GANGL

UNCLASSIFIED

AGARD-AR-90

NL

3 OF 6
ADA
044 915



CHAPTER 3. DATA DECLARATIONS

Data declarations are used to name and describe the data on which a program is to operate. All data in J73 must be declared before referencing.

J73 supports three basic data structures. The first and simplest is the scalar variable or item. Another structure is the table in which several items may be grouped together as an entry and dimensioned. Tables may have one to seven dimensions. The final data structure is a block. Scalar items, tables, and blocks may be grouped in blocks. Allocation order of items, tables, and blocks may be arranged by the overlay declaration.

$$\text{data declaration} ::= \left\{ \begin{array}{l} \text{item declaration} \\ \text{table declaration} \\ \text{block declaration} \end{array} \right\}$$

DATA ALLOCATION

Data may be allocated to one of three levels. The primary and most permanent data is reserve data. Unless otherwise indicated, data level defaults to reserve. Reserve data may be considered statically allocated. Only those values that are left in reserve data upon exit from a procedure are guaranteed at reentrance to the procedure.

The next level of data is that associated with a procedure. Procedure data values are not valid after exit from the containing procedure. Procedure data is further discussed in Chapter 6.

The final level of data is based data. No storage is allocated for based data by the compiler. Based data describes a structuring of data, a template if you will, which may be relocated dynamically. This relocation may be performed by declaring a default base — an item whose value is to be used as the address — or at each reference by using a formula whose value is the address.

The allocation level to be ascribed to data is indicated by an allocation specifier in the declaration.

$$\text{allocation specifier} ::= \left\{ \begin{array}{l} \text{IN} \\ \text{RESERVE} \\ @ \left\{ \begin{array}{l} \text{base} \\ \mu \end{array} \right\} \end{array} \right\}$$

$$\text{base} ::= \text{item name}$$

The presence of IN denotes that the data being declared is to be allocated local to the containing procedure. RESERVE data is permanently allocated. The presence of an @ indicates that the data is to be based. Based data may be declared with a default base, a scalar item name whose value is to be used as the address of the data. An item used as a base may also be declared based only if a default base is specified. Based data that have no default base must always be referenced with an explicit base formula. A base formula used at the reference to a based datum overrides any default base. Usage of based data is described in Chapter 4.

An allocation specifier is not permitted for data declarations within a block. Any allocation specifier desired must be placed on the containing block. Allocation specifiers are permitted for items and tables which are not declared in a block.

Blocks and tables that are formal parameters may not be declared with an allocation specifier; items that are formal parameters, if declared based, must include an implicit base.

ITEM DESCRIPTIONS

The basic ingredient of a data declaration is the description of the type, size, etc., of values that the data structure may take on. The item description is used to express these data attributes.

$$\text{item description} ::= \left\{ \begin{array}{l} C \left\{ \frac{\text{size specifier}}{\mu} \right\} \\ F \left\{ \frac{\text{significand specifier}}{\mu} \right\} \left\{ \frac{\text{exrad specifier}}{\mu} \right\} \\ \left\{ \frac{S}{U} \right\} \left\{ \frac{\text{size specifier}}{\mu} \right\} \left\{ \frac{\text{status list name}}{\mu} \right\} \end{array} \right\}$$

size specifier ::= number

significand specifier ::= number

exrad specifier ::= number

The abbreviations denote the type of item;

- C Indicates type character values. The length in bytes is defined by the size specifier. If the size specifier is omitted, the size is one character.
- F Indicates floating point. The significand specifier indicates the size in bits of the accuracy required. The default size is single word. The compiler will select the precision necessary. The exrad specifier indicates the required size in bits of the exrad. For determining this value, the radix is assumed to be two and the significand is assumed to be normalized to a fraction whose magnitude is greater than or equal to .5 and less than 1.0.
- S,U Indicates signed or unsigned integer. The size specifier indicates the size in bits required for the maximum magnitude. An additional bit is required to contain the sign. The maximum size signed integer is word size minus one. An unsigned integer may be specified as full word. The default size is the system dependent size required for addresses (one less for signed items). Both signed and unsigned integer items may be declared with a status list name. The presence of the status list name allows the data to be used in conjunction with status constants which are not qualified. An unsigned integer description may be used to obtain values to be used in bit formulas.

ITEM DECLARATIONS

An item declaration is used to define scalar data.

$$\begin{array}{l} \text{item declaration} ::= \text{ITEM } \text{item name} \rightarrow \\ \rightarrow \left\{ \frac{\text{allocation specifier}}{\mu} \right\} \text{item description} \left\{ = \left\{ \begin{array}{c} + \\ - \\ \mu \end{array} \right\} \frac{\text{constant}}{\mu} \right\}; \\ \text{item name} ::= \text{name} \end{array}$$

The allocation specifier is not permitted if the item declaration is within a block declaration. An item occupies as many whole words of space as is necessary to contain it. An optionally signed constant following an equal sign is used to preset the item with an initial value. If the type of the constant does not match the item, an appropriate conversion will be performed on the constant as for assignment (see CONVERSIONS). Presetting is allowed only if the item is allocated in reserve.

Items which are formal parameters may be declared based but, if so, must have an implicit base.

The following examples illustrate item declarations:

```
ITEM WEAPON U 3 WEAPON'TYP=V (CANNON'50MM);
ITEM MESSAGE C 72 = '*** UNEXPECTED EOF';
ITEM ADRES IN U;
ITEM LATITUDE @ ADRES F 15, 4;
ITEM LETTER C = 'M';
ITEM DEGREES RESERVE S 11 = -30.6;
```

WEAPON is declared as a 3-bit unsigned integer with an associated status list named WEAPON'TYPE and is preset to a status constant value which must be in the indicated status list since the constant is not otherwise qualified.

MESSAGE is a 72-character item with a 17-character preset; the remaining rightmost 55 characters will be filled with blanks.

ADRES is an unsigned item of default size whose residence level will be the procedure within which it is contained.

LATITUDE is a based item whose type is floating. Its precision is 15 bits and its maximum absolute value must be less than 16 or 2⁴. Unless overridden at its reference by an explicit base formula, the value of ADRES will be used to address LATITUDE.

LETTER is a one-character item preset to the character M.

DEGREES is an 11-bit signed integer declared in reserve. After the appropriate conversion, DEGREES will be preset to -30.

The allocation of WEAPON, MESSAGE, and LETTER, since not otherwise indicated, will be reserve data.

TABLE DECLARATIONS

There are two forms of table declarations: specified tables and ordinary tables. A specified table is declared such that entry size and table item position within the entry is completely specified. The compiler is responsible for arranging ordinary table entries. Both kinds of tables may be declared with from one to seven dimensions.

Tables may be declared with an item description or with a table body. A table may not have both an item description and a table body. The table body will contain the declarations of any table items which are part of the table. If the table is not declared with an item description, the table entry is given a default type of bit with a size specified as the number of words per entry times bits per word.

Status lists and defines may be declared within a table body. No special meaning is ascribed to this placement; it is simply permitted for convenience.

Table Dimensions

A table may have one to seven dimensions; the range of each dimension is described by a dimension list.

$$\begin{aligned} \text{dimension list} &::= [\left\{ \frac{\text{lower}}{\mu} : \text{upper} \right\} \sim] \\ \text{lower} &::= \left\{ \begin{array}{c} + \\ - \\ \mu \end{array} \right\} \text{integer} \\ \text{upper} &::= \left\{ \begin{array}{c} + \\ - \\ \mu \end{array} \right\} \text{integer} \end{aligned}$$

Lower must not be greater than upper. The length of any dimension is upper minus lower plus one. If the lower bound of a dimension is omitted, it is assumed to be zero. Upper and lower provide an inclusive range of a dimension index. The total number of entries in a table is the product of the length of all its dimensions. The dimension list

[1:7, 14, -6:4]

specifies a table with 7 times 15 times 11 or 1155 entries. Entries are allocated such that the rightmost dimension varies most rapidly, similar to the units digits in numerical notation. Although based tables cause no space to be allocated, the dimension list must be specified to insure proper addressing and indexing computations.

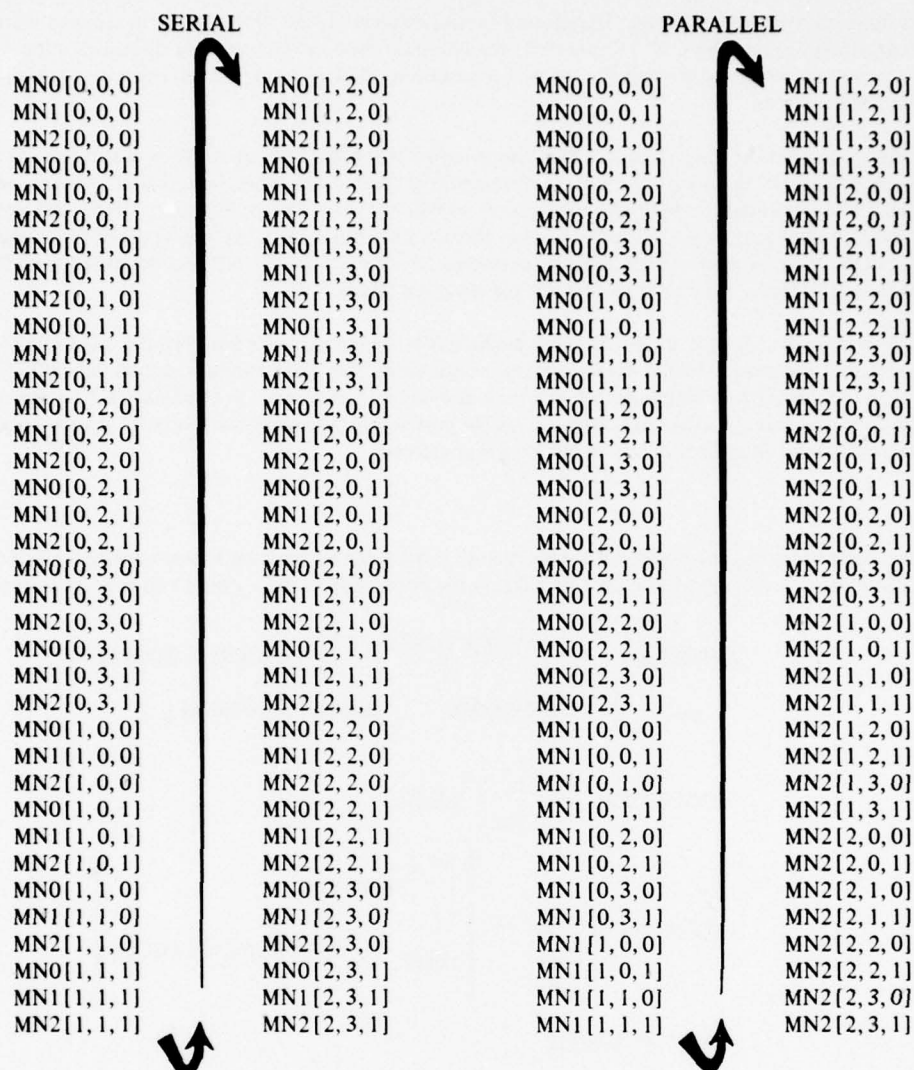
Table Structure

Tables may be of parallel or serial structure. The presence of the structure specifier indicates a parallel table; its absence indicates default to serial.

structure ::= P

Parallel structured tables allocate the first word (word zero) of each entry consecutively, followed by word one of all entries, then word two of all entries, etc. For serial tables, all words of entry zero are allocated together, then all words of entry one, etc. With the above structure in mind, the entry words are allocated in the order: all elements of a row — the rightmost dimension, all rows of a plane — the next to last dimension, next the planes of a volume, and so on.

The following example will illustrate both allocation order and parallel versus serial structure. A table MN with the dimensions, [2,3,1], and three words per entry, will be allocated as follows (MN0 is used as word 0, MN1 is word 1, MN2 is word 2).



The dimension indices are linearized at reference time by multiplying each index by the product of the dimensions to its right in the dimension list and summing the results, i.e., a reference to the previous dimension list with [i,j,k] results in an index expression of

$$i * 4 * 2 + j * 2 + k$$

This expression must be further multiplied by the words per entry for serial tables if greater than one and the address units per word (normally one on most machines). Notice then that serial table referencing is usually less efficient. Serial tables do, however, offer a more logical structure and are necessary for variable length entries.

The location (address) of a parallel table item is the location of the table plus the product of the table item's word of entry and the number of entries in the table. This is the first occurrence of the table item; subsequent occurrences are found by adding the indices. The location of a serial table item is the location of the table plus the table item's word of entry.

The use of multiple word entries in the context of assignment and relationals is usually more efficient for a serial table since all words of the entry are consecutive.

Table Packing

The allocation of table items to table entries is known as the packing of the item. The density of the packing desired for ordinary tables, or the packing resulting from a specified table item, is indicated by the following

$$\text{packing} ::= \begin{Bmatrix} N \\ M \\ D \end{Bmatrix}$$

N means full word usage, no packing. In ordinary tables, items are to be allocated the number of words required to contain them; for specified tables, N indicates that the compiler may assume no other item shares the word(s) which contain the item and accessing and storing may be performed on the full word(s). Unless otherwise specified, no packing is assumed in ordinary tables.

D indicates dense packing. In ordinary tables, the compiler will pack dense items such that items are allocated to the bits necessary to contain them with the following exceptions: only items whose size is greater than a word are allocated across a word boundary; a byte of a character item will not cross a word boundary; items will not overlap. In a specified table, packing density of D indicates that the compiler must not make assumptions regarding the bits adjacent to the item. A dense packed item must be extracted to remove adjacent bits and deposited such that adjacent bits remain unchanged. Dense packing is the default for specified tables.

Medium packing is specified by an M. Medium packing offers a compromise between the most efficient use of data space, dense, and the most efficient data accessing, no packing. Medium packing is system dependent but is the allocation of items to machine accessible part words such as bytes or half words. For ordinary tables, the compiler will allocate a medium packed item to the next larger accessible part word. For specified tables, M simply indicates that no other items exist in the medium field in which the item is contained.

Constant List

Tables and their items may be preset with initial values if the table is allocated as reserve data and if the table is not a formal parameter. A constant list is used to indicate the entry indices to be preset and the initial values desired.

$$\begin{aligned} \text{constant list} &::= \left\{ \left[\frac{\text{constant index}}{\mu} \sim \right] \right\} \text{constant list element} \rightarrow \\ &\rightarrow \left\{ \left[\frac{\text{constant index}}{\mu} \sim \right] \frac{\text{constant list element}}{\mu} \right\} \tilde{\beta} \\ \text{constant index} &::= \left\{ \begin{array}{c} + \\ - \\ \mu \end{array} \right\} \text{integer} \\ \text{constant list element} &::= \left\{ \begin{array}{c} \left\{ \begin{array}{c} + \\ - \\ \mu \end{array} \right\} \text{constant} \\ \text{count} \left(\frac{\text{constant list element}}{\mu} \right) \end{array} \right\} \sim \\ \text{count} &::= \text{number} \end{aligned}$$

A constant list consists of optionally signed constants separated by commas and indices. The initial indices indicate the starting entry of the following constant list element. If the initial indices are omitted, an index specifying the lower bound of each dimension is assumed. The indices are incremented in allocation order for every position in the constant list element; a position is separated by a comma. There is one more position than there are commas. Repeated positions may be indicated by a count preceding a parentheses enclosed constant list element. Null parentheses and successive commas indicate a skipped entry position. Repeated constant list elements may be nested. For instance, the constant list element

3.0,2(),3(1.0,2(,2.0))

is equivalent to

3.0,,,1.0,,2.0,,2.0,1.0,,2.0,,2.0,1.0,,2.0,,2.0

The occurrence of subsequent indices implies resetting. Indices may reset either backward or forward but the result of overlapping is undefined. Given a table whose dimension list is [-2:0,3], the following legal constant lists yield identical results.

[-1,2]'A', 'B', 'C', 'D'
[-1,2]'A', 'B'[0,0]'C', 'D'
[0,1]'D'[-1,3]'B', 'C'[-1,2]'A'

The constant index must specify a legal entry for the dimension; values beyond the range for a dimension are illegal. If the initial values specified do not correspond in type to the data being preset, the value will be converted accordingly as if the assignment were dynamic (see ASSIGNMENT STATEMENT).

Entries skipped have undefined values as do the remaining bits of a word partially preset. The presetting of the same entries of two overlapping items yields undefined results for the overlapped bits. Any attempt to preset the same

word as a result of overlaid data is also undefined. The presetting of interspersed words by separate data declarations and separate part words by multiple table item declarations that belong to the same table is legal and yields the expected and defined results.

Ordinary Table Declaration

Ordinary tables are used when particular table item position is not required. Ordinary tables are more machine independent than specified tables and with few exceptions are packed as well as the programmer arranged specified table.

$$\begin{aligned}
 \text{ordinary table declaration} &::= \text{TABLE } \text{table name} \left\{ \frac{\text{allocation specifier}}{\mu} \right\} \Rightarrow \\
 &\Rightarrow \text{dimension list} \left\{ \frac{\text{structure}}{\mu} \right\} \left\{ \frac{\text{packing}}{\mu} \right\} \Rightarrow \\
 &\Rightarrow \left\{ \begin{array}{l} \text{item description} \left\{ = \frac{\text{constant list}}{\mu} \right\} ; \\ \left\{ = \frac{\text{constant list}}{\mu} \right\} ; \text{ordinary table body} \end{array} \right\} \\
 \text{table name} &::= \text{name} \\
 \text{ordinary table body} &::= \left\{ \begin{array}{l} ; \\ \text{ordinary table item declaration} \\ \text{BEGIN} \left\{ \begin{array}{l} \text{ordinary table item declaration} \\ \text{status list declaration} \\ \frac{\text{define declaration}}{\mu} \end{array} \right\} \tilde{\beta} \text{ END} \end{array} \right\}
 \end{aligned}$$

The allocation specifier is not permitted if the table is declared within a block or is a formal parameter.

The packing specified in the table declaration applies to the item description following or the table items unless overridden individually in the table item declaration.

The constant list in the table declaration is converted as necessary to the type of the table and applies to the whole entry if an item description is omitted.

$$\begin{aligned}
 \text{ordinary table item declaration} &::= \text{ITEM } \text{table item name} \Rightarrow \\
 &\Rightarrow \text{item description} \left\{ \frac{\text{packing}}{\mu} \right\} \left\{ = \frac{\text{constant list}}{\mu} \right\} ; \\
 \text{table item name} &::= \text{name}
 \end{aligned}$$

Ordinary tables declared with an item description or a single table item declaration will result in an entry size equal to the number of words necessary to contain the item value. If no items are declared in the table, the entry size defaults to one word. In the remaining cases, the table items are allocated according to their individual packing specifier, or, if omitted, to that of the containing table. If the packing specifier is omitted from both the table item and the table declaration, no packing is assumed. The order of table item declarations within a table body implies no allocation order; the compiler is free to rearrange the table items to optimize the packing. The entry size of the table is the total number of words required to contain all items as allocated.

Some examples of ordinary tables are illustrated below:

```

TABLE ARRAY IN [1:4, 1:5] F;
TABLE FLIGHT'PLAN @ [0]; BEGIN
  ITEM FLIGHT'NO U 5 D;
  STATUS LIST V(PRIVATE), V(COM'L), V(MILTRY), V(UFO);
  ITEM AIRCRAFT'CATEGORY U 2 LIST M;
  ITEM ALTITUDE F;
  ITEM DESTINATION C 25;
  ITEM CREW'SIZE U 4; END
TABLE SPACE RESERVE [5000:10000] = 5001(0);
TABLE TAB [-3:-1,14] M; BEGIN
  ITEM ABC S 17 = 20(1);
  ITEM DFG S 5 = 20(2,1);
  ITEM GHI C D = 'A',5(' '),'$';
  ITEM JKL U 5 D; END

```


ARRAY is a local 4x5 table of floating values. FLIGHT'PLAN is a based table. SPACE is a one-word-per-entry reserve table initialized to zeroes. TAB is a 3x15 table whose residence level is reserve; TAB has preset values.

Specified Table Declaration

A specified table is used when a particular item arrangement is required. The requirement may be imposed by an external interface, a need for tighter packing than the compiler produces (this should be unusual), specific item overlapping, or because of a requirement for variable length entries. The entry size and table item position are completely described in a specified table.

$$\begin{aligned}
 \text{specified table declaration} &::= \text{TABLE } \text{table name} \rightarrow \\
 &\rightarrow \left\{ \frac{\text{allocation specifier}}{\mu} \right\} \text{dimension list} \left\{ \frac{\text{structure}}{\mu} \right\} \text{words per entry} \rightarrow \\
 &\rightarrow \left\{ \begin{array}{l} \text{item description} \left\{ \frac{\text{packing}}{\mu} \right\} [\text{first bit} \left\{ \frac{\text{word number}}{\mu} \right\}] \left\{ = \frac{\text{constant list}}{\mu} \right\} ; \\ \left\{ \frac{\text{constant list}}{\mu} \right\} ; \text{specified table body} \end{array} \right\} \\
 \text{specified table body} &::= \left\{ \begin{array}{l} ; \text{specified table item declaration} \\ \text{BEGIN} \left\{ \begin{array}{l} \text{specified table item declaration} \\ \text{status list declarations} \\ \text{define declaration} \\ \mu \end{array} \right\} \tilde{\beta} \text{ END} \end{array} \right\} \\
 \text{words per entry} &::= \text{number} \\
 \text{first bit} &::= \text{number} \\
 \text{word number} &::= \text{number}
 \end{aligned}$$

The allocation specifier, dimension list, structure, and constant list are treated as in an ordinary table. The allocation specifier is not permitted if the table is in a block.

The words per entry specifier indicates that the table is specified and denotes the entry size. Indexing into a specified table uses the declared words per entry. If an item description is present, it defines the type of values associated with the table name. The first bit defines the starting bit position of the value; word number specifies the word of entry in which the item begins. Both bit number and word number count from zero. Word number is assumed to be 0 if omitted. The packing in the table declaration applies to the item description and describes the accessing to be used for references to the table name value. If the packing specifier is omitted, dense packing is assumed.

If the item description is omitted from the table declaration, the table is type bit and has a size of words per entry times bits per word.

$$\begin{aligned}
 \text{specified table item declaration} &::= \text{ITEM } \text{table item name} \rightarrow \\
 &\rightarrow \text{item description} \left\{ \frac{\text{packing}}{\mu} \right\} [\text{first bit} \left\{ \frac{\text{word number}}{\mu} \right\}] \rightarrow \\
 &\rightarrow \left\{ = \frac{\text{constant list}}{\mu} \right\} ;
 \end{aligned}$$

First bit indicates the starting bit of the table item. The packing specifier describes the accessing that results from the specification; packing defaults to dense if unspecified.

Word number indicates the word of entry in which the item begins. Word number defaults to zero. Word number of a table item may be specified beyond the entry size of the table. This specification results in a very useful structure, namely the concept of variable length entries.

Therefore a logical entry may be created which may be considered as a based entry within the table; the indices simply locates the top of the entry. Notice that this concept is only feasible for serial tables since any table items declared beyond the entry size in a parallel table would have an initial location outside of the table. Searching a table of this nature may be performed by having the entries linked together in some fashion, the link or indices for each subsequent entry might exist in the prior entry. Notice that entries need not be linked in allocation order. Searching variable length entry tables may also be done using an algorithm which steps from entry to entry being cognizant of the logical entries format and its corresponding size.

The following examples will illustrate specified tables and the concept of variable length entries.

```

TABLE BLIPS [100] P 3;
BEGIN
    ITEM XCOORD S 15 M [2];
    ITEM YCOORD S 15 M [2,1];
    ITEM LATITUDE U 7 [26,1];
    ITEM SPEED U 10 [18,0];
    ITEM FRIENDLY U 1 [18,1];
    ITEM TYP C 4 N [0,2];
END

TABLE INVENTORY [500] 1;
BEGIN
    ITEM PART C 4 N [0]='NUT',,,2('BOLT', 4()),'ASSY';
    ITEM ON'ORDER U 1 D [31,1]=0 [3]1,4(),1 [13]0;
    ITEM NXT'PRT U 16 M [0,1]=3,,,8 [8]13 [13]0;
    ITEM NO'PARTS U 8 D [16,1]=[13]25 [8]2 [0]4,,,13;
    STATUS HD V(PHILIPS), V(REG), V(ALLEN);
    ITEM HEAD U 2 HD N [30,4]=[3]V(ALLEN) [8]V(PHILIPS);
    ITEM LENGTH F N [0,3]=[3]3.25,4(),1.75;
    ITEM SIZ F N [0,2]=.375,,,1.5 [8]1.25E-1;
    ITEM LOCK U 1 [0,3];
END

```

The first table is a 101 entry, one dimensional, parallel, three words per entry table. Its logical entry on a 36-bit machine would appear as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35			
XCOORD																		SPEED																		WORD 0		
YCOORD																				FRIENDLY						ALTITUDE										WORD 1		
TYP																																						WORD 2

Table INVENTORY is a variable entry table whose entry structure depends on the part type. In this example, an ASSY entry is two words, a BOLT entry is five, the NUT entry is three, and WASHER entries are four words. NXT'PRT is used to locate the next entry. The layout and values of INVENTORY on a 32-bit machine are shown on the next page (double lines denote separation of logical entries).

BLOCK DECLARATIONS

Blocks are used to group items, tables, and other blocks. They may be used as a structure of data that is common to several programs. Blocks may also be used as a logical record to be referenced in an I/O procedure call. Based blocks can be used to overlay data attributes onto dynamic space.

LOGICAL
ENTRY

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PART = 'NUT'																															
NXT'PRT = 3																NO'PARTS = 4								ON'ORDER → 0							
SIZ = 0.375																															
PART = 'BOLT'																															
NXT'PRT = 8																NO'PARTS = 13								ON'ORDER → 1							
SIZ = 1.5																															
LENGTH = 3.25																															
																								HEAD = V(ALLEN) →							
PART = 'BOLT'																															
NXT'PRT = 13																NO'PARTS = 2								ON'ORDER → 1							
SIZ = 1.25 E-1																															
LENGTH = 1.75																															
																								HEAD = V(PHILIPS) →							
PART = 'ASSY'																															
NXT'PRT = 0																NO'PARTS = 25								ON'ORDER → 0							

block declaration ::= BLOCK block name $\left\{ \begin{array}{c} \text{allocation specifier} \\ \mu \end{array} \right\} ; \blacktriangleright$

$\blacktriangleright \left\{ \begin{array}{l} \text{data declaration} \\ \text{BEGIN} \left\{ \begin{array}{l} \text{data declaration} \\ \text{overlay declaration} \\ \text{status declaration} \\ \text{define declaration} \\ \mu \end{array} \right\} \tilde{\beta} \text{ END} \end{array} \right\}$

block name ::= name

A block declaration within a block or a formal parameter block may not contain an allocation specifier. No allocation order is implied by the order of the declarations within a block. If a particular allocation order is required, an overlay declaration should be used. Items and tables within a block may not have initial values unless the block is reserve data.

Status list and define declarations may appear in a block declaration for convenience. No special significance is associated with this placement.

The size of a block is the number of words required to contain the data within. If no data is declared within a block, its size is zero.

Blocks have little use in J73 statements except that a block may be used as a procedure parameter. The primary utility of a block is simply as an aggregate of data.

If a block is declared based, the data within the block is also based and may be considered as having a relative address, namely, the position of the data within the block or the offset from the top of the block. This relative offset is applied to any implicit or explicit base for the block to locate the block's data.

A sample block declaration is shown below.

```
BLOCK WORKAREA;
BEGIN
  ITEM TABX U 6; "TAB CONTROL WORD"
  TABLE TAB [0:31] P 2;
  BEGIN
    ITEM T1 F N [0];
    ITEM T2 C 5 [0,1];
  END
  TABLE ERMSGs[1:9] 4 C 20 [0] =
    "1" 'INVALID HEADER',
    "2" 'END OF FILE',
    "3" 'PARITY ERROR',
    "4" 'END OF REEL',
    "5" 'RECORD FORMAT ERROR',
    "6" 'NO EOF',
    "7" 'TAPE IS 556 BPI',
    "8" 'LOAD NEXT REEL',
    "9" 'TAPE UPSIDE DOWN';
END
```

Block WORKAREA is allocated to reserve space and has a size of 101 words, $1 + 2 \times 32 + 4 \times 9$. However, the order of TABX, TAB, and ERMSGs within WORKAREA is system dependent.

OVERLAY DECLARATION

The overlay declaration is used to arrange the order of data allocation, to overlay data for time-sharing of core or to establish equivalence, and to force the location of data to a specific address.

```
overlay declaration ::= ➤
➤ OVERLAY { { {  $\frac{\text{number}}{\text{bit constant}}$  } } } }  $\frac{\text{overlay expression}}{\mu}$  ;

overlay expression ::= overlay string ~
overlay string ::= overlay element ~
overlay element ::= {  $\frac{\text{spacer}}{\text{data name}}$  }
                    { ( overlay expression ) }
spacer ::= number
data name ::= {  $\frac{\text{item name}}{\text{table name}}$  }
              {  $\frac{\text{block name}}$  }
```

Overlay strings are used to arrange overlay elements consecutively; the length of an overlay string is the sum of the lengths of the individual overlay elements. A spacer is used to indicate a spacing of a number of words. The overlay

```
OVERLAY AA, BB, 10, CC;
```

will allocate BB immediately following AA, and CC 10 words after the end of BB.

Overlay expressions are used to allocate overlay strings such that they begin at the same location. The length of an overlay expression is the length of the longest overlay string. The length of the following overlay expression

AA, 5 : BB, 10 : 40

would be the longest of the length of AA + 5, the length of BB + 10 or 40. AA and BB would begin at the same address. Overlay expressions may be enclosed in parentheses and used as an overlay element in subsequent overlay strings.

The optional bracketed value may be used to specify a particular address for the initial overlay element of the overlay.

Given the following declarations:

```

ITEM S1 S;
TABLE T100[99];
BLOCK B30;
BEGIN
    ITEM F1 F;
    TABLE T29[30:58] F;
END
ITEM U1 U;
TABLE T34[1:17] N;
BEGIN
    ITEM C34 C;
    ITEM S34 S 2;
END
OVERLAY [ 1000 ] B30,(T100:50,(S1:T34)),U1;
```

on a word addressed computer, the decimal addresses of the overlaid elements would be (notice that the names of the data in the example include their lengths in words):

```

B30   1000
T100  1030
S1    1080
T34   1080
U1    1130
```

Overlays may be used to relate program data to data external to the program by including amongst the overlay elements a reference to compool or external data (to be described later) or by using the facility to force an overlay to a specific address. Data may appear in more than one overlay thereby establishing a relative correspondence between the overlays. In all cases, it is the programmer's responsibility to prevent any contradictions of allocation.

An overlay declaration within a block is restricted to arranging data declared within that block. Data declared based may not be overlaid except that data within a based block may be overlaid to establish allocation order within the block. An overlay declaration within a block may not contain an address specification.

All data referenced in an overlay declaration must be declared prior to the overlay reference.

All data of an overlay are allocated at the same residence level; all in reserve or all in one particular procedure. Each overlay element must be declared explicitly or by default to have the same residence level as every other overlay element within the overlay.

CHAPTER 4. FORMULAS

Formulas describe the algorithms to be applied to data operands to compute values. In general, J73 permits the application of all operations to all data types. If the operand type is not appropriate, an implicit conversion is applied to the operand. Throughout the specification where a particular formula or operand type is indicated, a formula of another type may be substituted; any necessary conversion will be supplied. These conversions are described later in this chapter.

Formulas comprise one or more of the three primary operand forms modified or combined by some operator. The primary operands are constants, variables, and function results. The operators are addressing, numeric, relational, and logical. The addressing operators are indexing and the @ (basing) operator. Both will be described in the section on variables. The remaining operators will be described later in this chapter.

VARIABLES

Variables are the J73 data in which values from computations are maintained for subsequent reference. Variables may be scalar data, dimensioned or indexed data, and functional data.

$$\begin{aligned}\text{variable} &::= \left\{ \begin{array}{l} \text{named variable} \\ \text{function variable} \end{array} \right\} \\ \text{named variable} &::= \left\{ \begin{array}{l} \text{scalar variable} \\ \text{indexed variable} \end{array} \right\} \\ \text{scalar variable} &::= \text{item name} \left\{ @ \frac{\text{base formula}}{\mu} \right\} \\ \text{base formula} &::= \text{numeric formula}\end{aligned}$$

A scalar variable which is based may be referenced with an explicit base or with the default base from its declaration. The value of either base will be converted to integer as necessary. The value of the base will be expressed in address units.

$$\begin{aligned}\text{indexed variable} &::= \left\{ \begin{array}{l} \text{table name} \\ \text{table item name} \end{array} \right\} [\text{index} \sim] \left\{ @ \frac{\text{base formula}}{\mu} \right\} \\ \text{index} &::= \text{numeric formula}\end{aligned}$$

An indexed variable is used to select a particular occurrence or entry of a table or table item. An index must be present for every dimension of the table. Indices and base formulas will be converted to integer as necessary. An index may not yield a value that is beyond the range of the corresponding dimension. The indices are linearized as described in Chapter 3 on tables, converted to address units, and added to the address of the table or table item to locate the beginning of the desired entry. The example below illustrates this computation.

Based Data

Every based datum referenced must have been declared with a default base or be referenced with the @, read at, operator and a base formula. Except for the purposes of this description, a base formula is no more nor less than a numeric formula. When a based variable is referenced with an explicit base formula, any default base is overridden. In either case, the value of the base or base formula is to be used as the address (in address units) of the data structure declared as based. The location of based data such as table items within a based table or other data within a based block is determined by adding the relative address of the referenced datum to the address (base value) of the containing structure. Any indices which are present are applied to this total address. Given the following declarations:

```
BLOCK AA@; BEGIN
  ITEM BB F;
  TABLE CC[1] 3;
  ITEM DD F [0,1];
OVERLAY BB, CC; END
```

A reference to DD[1]@10 is computed by adding the value of the base formula, 10, to the relative address of DD within AA, 2, and then adding the index (which has an implied multiplier of three because of its serial structure). The final address of DD then is 10+2+3x1 or 15. The above description assumes a machine whose address units are in terms of words. On a byte addressed machine with four bytes per word, the address of DD would be 10+2x4+3x1x4 or 30.

Only based data may be legally referenced using the @ operator.

BIT(S5[1],0,11-2 "REFERS TO BITS 30-31 IN WORD 3 OF ENTRY 1"
 BIT(C16[11+JJ],KK*6-4,5) "REFERS TO BITS 33-34 OF WORD 1 AND BITS 0-2
 OF WORD 2 OF ENTRY 1"
 BYTE(C16[1],4,KK) "REFERS TO THE BYTES OF ENTRY 1 CONTAINING
 THE CHARACTERS 'EFGHIJKL' "

Variable Types

Variables have type and size. The attributes of a function variable are described in the previous section. Named variables assimilate the attributes of the item description associated with the data name. The type of a table name declared without an item description is bit; its size is the total number of bits in all words of an entry.

$$\begin{aligned} \text{character variable} &::= \left\{ \begin{array}{l} \text{named variable} \\ \text{byte function variable} \end{array} \right\} \\ \text{bit variable} &::= \left\{ \begin{array}{l} \text{table name [index } \sim \text{] } \left\{ \begin{array}{l} @ \text{ base formula} \\ \mu \end{array} \right\} \\ \text{bit function variable} \end{array} \right\} \end{aligned}$$

A bit variable may only be a table name variable if the table is declared without an item description.

$$\begin{aligned} \text{numeric variable} &::= \left\{ \begin{array}{l} \text{integer variable} \\ \text{floating variable} \end{array} \right\} \\ \text{integer variable} &::= \text{named variable} \\ \text{floating variable} &::= \text{named variable} \end{aligned}$$

In general, a variable of any type may be used in place of any other variable. If a specific type is called for, the compiler will supply the appropriate conversions. The rules for conversion are described later in this chapter.

Some sample variables are:

FLIGHT [4] "ENTRY 4 OF VARIABLE FLIGHT"
 BIT (VECTOR[LEFT-6,RIGHT+4],0) "BIT 0 OF THE INDICATED ENTRY OF
 VECTOR"
 TAB [11]@TAB'ADR "ENTRY 11 OF TAB BASED AT THE ADDRESS
 SPECIFIED BY THE VALUE OF TAB'ADR"

FUNCTIONS

A function is a procedure declared with an item description that specifies the result of the procedure. Certain functions are known intrinsically to the language; these intrinsic functions are described in the next section.

$$\begin{aligned} \text{function call} &::= \left\{ \begin{array}{l} \text{intrinsic function call} \\ \text{procedure name } \left\{ \begin{array}{l} @ \text{ data base} \\ \mu \end{array} \right\} \rightarrow \\ \rightarrow (\left\{ \begin{array}{l} \text{actual input parameter} \\ \mu \end{array} \right\} \sim) \end{array} \right\} \\ \text{data base} &::= \text{numeric formula} \end{aligned}$$

A function call is the process of assigning the actual input parameters to the formula input parameters of the procedure named, execution of the named procedure, and lastly, substitution of the function result value into the formula in which the function call appeared.

The procedure named in a function call must be a procedure declared with an item description that specifies the attributes of the resulting value.

If the procedure is declared as based, a numeric formula must be included in the call to provide the address of the procedure's data base.

Functions will be discussed further with procedure calls and procedure declarations.

Intrinsic Functions

Intrinsic functions may be thought of as a multiple operand operator that is expressed in functional notation rather than as the more usual infix notation.

$$\text{intrinsic function call} ::= \left\{ \begin{array}{l} \text{bit function call} \\ \text{byte function call} \\ \text{loc function call} \\ \text{shift function call} \\ \text{abs function call} \\ \text{sign function call} \\ \text{size function call} \\ \text{data size function call} \end{array} \right\}$$

Bit Function

The bit function is an extension of the bit function variable. Whereas a bit function variable may only be applied to a named variable, the bit function may be applied to any formula. However, the bit function may not be used in a context requiring an address rather than a value such as in assignments and as actual output parameters.

$$\begin{aligned} \text{bit function call} &::= \text{BIT} (\text{bitee} , \text{fbit} \left\{ , \frac{\text{nbit}}{\mu} \right\}) \\ \text{bitee} &::= \text{formula} \\ \text{fbit} &::= \text{numeric formula} \\ \text{nbit} &::= \text{numeric formula} \end{aligned}$$

The bit function yields a bit formula whose size is the size of the bitee (or one bit if nbit is omitted). The value of the bit function is the substring of nbit bits of the bitee beginning with the first bit indicated by fbit; this substring is right justified and padded with zeros as necessary to the size of result. If nbit is not specified, only the bit indicated by fbit is extracted. Bits are numbered from the left beginning at zero. If the value of nbit is zero, the bit function results in a value of 0. Fbit and nbit must not describe a substring beyond the extents of the bitee. Fbit and nbit will be converted to integer as necessary.

BIT(3B'76543',6,6) "YIELDS THE VALUE 54 (OCTAL)"

Byte Function

The byte function call yields a character formula whose value is a substring of the character value to which the function is applied. As with the bit function, a byte function may not be used in an assignment context while a byte function variable may.

$$\begin{aligned} \text{byte function call} &::= \text{BYTE} (\text{bytee} , \text{fbyte} \left\{ , \frac{\text{nbyte}}{\mu} \right\}) \\ \text{bytee} &::= \text{character formula} \\ \text{fbyte} &::= \text{numeric formula} \\ \text{nbyte} &::= \text{numeric formula} \end{aligned}$$

The byte function produces a result whose value is a substring of nbyte bytes of the bytee beginning with the byte designated by fbyte. Bytes are numbered left to right beginning with zero. If nbyte is omitted, it defaults to one byte. The size of the result is the size of the bytee (or one byte if nbyte is not indicated). If nbyte has a value of zero, the result is all blanks. The byte function is undefined if fbyte and nbyte describe a substring outside the bounds of the bytee.

BYTE(DAY'OF'WEEK('5/9/75'),0,2) "PRODUCES 1ST 2 CHARACTERS OF WEEK-DAY NAME"

Loc Function

The loc function may be applied to a program entity to obtain its machine address or location.

$$\text{loc function call} ::= \text{LOC} (\left\{ \begin{array}{l} \text{statement name} \\ \text{named variable} \\ \text{procedure name} \\ \left\{ \begin{array}{l} \text{table name} \\ \text{block name} \end{array} \right\} \left\{ @ \frac{\text{base formula}}{\mu} \right\} \end{array} \right\})$$

The loc function yields an unsigned integer value of default unsigned size. Its value is the address in address units of the first word of the indicated entity. Notice that since this location may be made up of a base formula plus a relative location from the top of the based structure, it is quite possible that LOC(AA@BB) is not equal to BB.

Abs Function

The abs function produces a numeric formula the same size and type as the numeric formula to which it is applied except if applied to a nonfloating formula the result is unsigned.

abs function call ::= ABS (numeric formula)

The value of the abs function is the absolute value of its parameter.

Sign Function

The sign function produces a signed one bit integer.

sign function call ::= SGN (numeric formula)

The value is defined as follows:

numeric formula	result
> 0	+1
= 0	0
< 0	-1

SGN (3.7) is equal to +1.

Shift Function

The shift function is a logical operation applied to a bit formula.

shift function call ::= SHIFT (shiftee , shift count)

shiftee ::= bit formula

shift count ::= numeric formula

The shift function produces a result of type bit whose size is the same as the shiftee. The value of the shift function is obtained by performing a logical shift of the shiftee by the number of positions indicated by the shift count. If the count is positive the shift is to the left; if negative, the shift is right. The shift function may be thought of as a logical shift being performed in a register the same size as the shiftee. Vacated bits are filled with zeros. Bits shifted off the end are lost. A shift count greater than the size of the shiftee yields an undefined result. A shift count equal to the size of the shiftee produces a value of zero.

Since the shift function is a logical operation, its usage is limited to the operands whose size does not exceed the maximum for unsigned integers.

SHIFT(3B'76543',-6)

The above shift function produces a result whose size is 15 bits and value is equal to 3B'00765'.

Size Function

J73 includes three functions which produce the size of data. The resultant value represents the size in bits, bytes or words as indicated.

size function call ::= $\left\{ \begin{array}{l} \text{BITSIZE} \\ \text{BYTESIZE} \\ \text{WORDSIZE} \end{array} \right\} \left(\left\{ \begin{array}{l} \text{formula} \\ \text{table name} \\ \text{block name} \end{array} \right\} \right)$

The size function call is an unsigned integer of default size. The value of the size function is the number of units (bits, bytes, or words) of the formula, table or block. The size of a table is all words of all entries expressed in appropriate units. Given the table

TABLE SIZ[10:15.6] P 2; ;

WORDSIZE (SIZ) equals 84.

Data Size Function

The data size function is applied to a procedure and produces an unsigned integer of default size whose value is the number of words of data required by the procedure.

data size function call ::= DSIZE (procedure name)

The data size function may only be applied to procedures declared as having based data space. The meaning and purpose of this function will be described with procedure declarations in Chapter 6.

FORMULAS

Formulas comprise three data types.

$$\text{formula} ::= \left\{ \begin{array}{l} \text{character formula} \\ \text{bit formula} \\ \text{numeric formula} \end{array} \right\}$$

The definition of each formula type is described in the ensuing sections.

CHARACTER FORMULAS

A character formula is a value composed of a string of characters. Its size is measured in bytes or number of characters.

$$\text{character formula} ::= \left\{ \begin{array}{l} \text{character constant} \\ \text{character variable} \\ \text{character function call} \\ (\text{character formula}) \end{array} \right\}$$

character function call ::= function call

A character function call is a function call to a procedure declared with a character item description.

BIT FORMULAS

A bit formula is a string of bits. Numeric and character formulas used in a context demanding a bit formula will be converted to type bit. Floating formulas will be converted first to integer; integer and character formulas will be simply treated as bits. Unused filler bits will be ignored.

$$\text{bit formula} ::= \left\{ \begin{array}{l} \text{bit constant} \\ \text{bit variable} \\ \text{bit function call} \\ \text{logical formula} \\ \text{relational formula} \\ (\text{bit formula}) \\ \text{numeric formula} \\ \text{character formula} \end{array} \right\}$$

Logical Formulas

A logical formula is a logical operator applied to a bit formula. Bit formulas used in logical formulas are restricted to the maximum unsigned integer size; any operand which is greater than this size is truncated on the left.

$$\text{logical formula} ::= \left\{ \begin{array}{l} \text{bit formula} \left\{ \begin{array}{l} \text{AND} \\ \text{OR} \\ \text{XOR} \\ \text{EQV} \end{array} \right\} \text{bit formula} \\ \text{NOT bit formula} \\ \text{shift function call} \end{array} \right\}$$

The logical operator NOT produces a result the same size as its operand with a value that is the logical complement of its operand.

AND, OR, XOR (exclusive OR), and EQV (equivalence) perform their usual logical function on a bit by bit basis. If, after any necessary truncation is effected to reduce the operand size to the maximum size, the operand sizes do not match, the shorter is padded on the left with zeros to the size of the longer. The size of the result is the size of the longer operand. The result of the formula

AA AND ((BB XOR CC) EQV (DD OR EE))

given the values

AA = 3B'17577'

BB = 3B'3543'

CC = 3B'27336'

DD = 3B'65620'

EE = 3B'53'

is the value 3B'16571'.

Relational Formulas

Relational formulas are formulas compared with each other using the relational operators.

$$\text{relational formula} ::= \text{formula} \left\{ \begin{array}{l} = \\ <> \\ <= \\ >= \\ < \\ > \end{array} \right\} \text{formula}$$

The result of a relational formula is a one bit bit formula whose value is 1 if the relation between the two formulas is as indicated by the relational operator. Otherwise the result is zero. If the formulas are of the same type the comparison is performed in that type. If the types of the formulas are of different type, a conversion is applied to the formula of lower type to the type of the other formula. The type hierarchy and conversion order is character to bit, bit to integer, and, finally, integer to floating.

When comparing two character formulas, the shorter is first padded on the right with blanks to the size of the longer.

For bit operands the shorter is padded on the left with zeros to the size of the longer.

Relational formulas combined by logical operators will be evaluated only so far as is necessary to determine the value. Given the values and formula below:

AA[BB] has the value V(GOOD)

CC has the value 'A'

DD has the value 3.6

(AA[BB] <= FF(0) OR CC > 'A') AND DD <> 5E-2

the relational CC > 'A' will not be evaluated if FF(0) yields a value of V(GOOD); conversely, DD <> 5E-2 will not be evaluated if FF(0) is less than V(GOOD) and CC is not greater than 'A'.

Conditional Formulas

A conditional formula is required in certain statement contexts after the primitives IF and WHILE and in the trace directive.

conditional formula ::= bit formula

The conditional formula produces a truth value for a bit formula by examining the rightmost bit. If the value of the bit is one the conditional formula is true; if the value is zero the conditional formula is false. Any non-bit formula used as a conditional formula will be converted to bit before testing.

NUMERIC FORMULAS

Numeric formulas specify arithmetic operations on numeric values.

$$\begin{aligned}
 \text{numeric formula} &::= \left\{ \begin{array}{l} \text{numeric constant} \\ \text{numeric variable} \\ \text{numeric function call} \\ \text{bit formula} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{ numeric formula} \\ \text{numeric formula} \left\{ \begin{array}{l} + \\ - \\ * \\ / \\ \backslash \\ ** \end{array} \right\} \text{ numeric formula} \\ (\text{ numeric formula }) \end{array} \right\} \\
 \text{numeric function call} &::= \left\{ \begin{array}{l} \text{floating function call} \\ \text{integer function call} \\ \text{status function call} \end{array} \right\} \\
 \text{floating function call} &::= \text{function call} \\
 \text{integer function call} &::= \text{function call} \\
 \text{status function call} &::= \text{function call}
 \end{aligned}$$

A bit formula used for a numeric formula is treated as an unsigned integer value. If the size of the bit formula exceeds that size permitted for numeric formulas, leading bits are truncated.

The binary numeric operators are performed in floating if either operand is floating. Exponentiation is performed in floating unless the base is non-floating, the exponent is a positive integer constant and the value of the exponent times the size of the base is less than or equal to the size of the largest integer value.

The size of the integer result of a binary operator is described below (floating results are all single precision). The integer bits of a value are referred to as I; S is the maximum size for a signed integer. A subscript will denote the value of concern.

1 is left operand
 2 is right operand
 N is the numerator
 D is denominator
 M is modulus
 B is the base of an exponentiation
 E is the exponent
 R is the result

For addition and subtraction

$$I_R = \text{minimum}(S, \text{maximum}(I_1, I_2) + 1)$$

If either operand is signed or the operation is subtraction, the result is signed.

For multiplication

$$I_R = \text{minimum}(S, I_1 + I_2)$$

The result is signed if either operand is signed.

For division

$$I_R = I_N$$

The result is signed if either operand is signed.

For modulus, the value is defined as

$$x \setminus y = x - y * \text{trunc}(x/y)$$

where $\text{trunc}(v)$ is an integer whose sign is the same as v and whose magnitude is the largest integer less than or equal to the absolute value of v . For y equal to zero, $x \setminus y$ is undefined. The resultant size of modulo is defined as

$$I_R = \text{minimum}(I_N, I_M)$$

The result is signed if either operand is signed.

For exponentiation,

$$I_R = I_B * \text{value}_E$$

The result is signed if the base is signed.

EVALUATION ORDER, PRECEDENCE

The order of operator-operand combination in a formula is determined by operator precedence and as grouped by parentheses. In the absence of parentheses, operands connected by operators of equal precedence are combined into formulas from left to right.

The precedence of the operators in J73 are

0	=					(assignment)
1	EQV		XOR			
2	OR					
3	AND					
4	NOT					
5	=	<>	<	<=	>=	>
7	+	-				
8	*	/	\			
9	**					
10	@		indexing		function call	

The evaluation order of formulas is unspecified except that base formulas and indices must be computed before reference to a variable. When a binary operator immediately precedes a unary operator, the unary operation is performed first.

CONVERSIONS

J73 permits free mixing of formula types. If the type of the formula supplied does not match the type of formula required, the compiler supplies the necessary conversions. The conversions proceed in hierarchical order either up to the type desired or down. The type hierarchy is character at the lowest level, followed by, in order, bit, unsigned integer, signed integer, then floating. The conversions are described below.

Character formulas which require size changes are adjusted by adding blank characters on the right or deleting trailing characters. If a conversion from character is required, the bits of the bytes are simply treated as type bit. Any unused bits in a character word are ignored. These bits may arise on machines whose word size is not an integral multiple of the bits per byte.

Bit formulas which require a size adjustment are padded with zeroes or truncated on the left to the required size. The conversion of bit to integer is essentially a null conversion except that if the bit formula is too long leading bits are truncated. Conversion of bit to character is similarly a treatment of the bit string as the bits of the bytes. Any filler bits required to fill a word after collecting a words worth of characters are undefined and do not derive from bits of the bit formula. For instance on a 36-bit computer with five 7-bit bytes per word, a 70-bit bit formula converts to 10 bytes of characters. The extra bit of each word is added in the conversion.

The conversion of integer formulas to bit formulas is simply a bit for bit substitution. No value or size change results. Conversion of unsigned integer values to signed integer values is simply the treatment of the unsigned value as a signed value and vice versa.

Conversion of integer formulas to floating is performed as appropriate to the object computer to the precision permitted in floating values. The conversion of floating to integer results in a truncation of a fractional value.

NUMBER FORMULAS

Within a J73 program a formula made up entirely of certain compile time values may be substituted for a number.

$$\begin{aligned} \text{number} &::= \left\{ \begin{array}{l} \text{digit} \sim \\ (\text{number formula}) \end{array} \right\} \\ \\ \text{number formula} &::= \left\{ \begin{array}{l} \begin{array}{l} \text{integer} \\ \left\{ \begin{array}{l} + \\ - \\ \text{NOT} \\ \mu \end{array} \right\} \text{number formula} \end{array} \\ \\ \begin{array}{l} \text{number formula} \left\{ \begin{array}{l} + \\ - \\ * \\ / \\ \backslash \\ = \\ <> \\ <= \\ >= \\ < \\ > \\ \text{AND} \\ \text{OR} \\ \text{XOR} \\ \text{EQV} \end{array} \right\} \text{number formula} \\ \\ \text{SHIFT}(\text{number formula}, \text{number formula}) \\ \text{ABS}(\text{number formula}) \\ \text{SGN}(\text{number formula}) \\ (\text{number formula}) \end{array} \right\} \\ \\ \text{integer} &::= \left\{ \begin{array}{l} \text{'character'} \\ \text{bit constant} \\ \text{number} \\ \text{qualified status constant} \end{array} \right\} \end{aligned}$$

The number formulas are restricted to operands of maximum integer size or less. The meanings and precedence of operators in a number formula is identical to the more general formulas. Note that multiple byte character constants, floating constants, and the exponentiation operator are excluded from number formulas.

EXAMPLE FORMULAS

Some examples of formulas are

SIN(ALPHA)**2+COS(ALPHA)**2<>1 "SHOULD ALWAYS BE FALSE"

LOWER<ALTITUDE[BLIP] AND ALTITUDE[BLIP]<UPPER "TRUE IF ALTITUDE
WITHIN LOWER AND
UPPER"

ENTRY[LINK[INDEX+4]*2,FUDGE]/10

(BITSIZE(TAB[COUNT])+WDSIZ-1)/WDSIZ "# WHOLE WORDS REQUIRED TO
CONTAIN TAB"

BITSIZE(TAB[COUNT])\WDSIZ "BITS LEFT IN PARTIAL WORD"

SHIFT(VECTOR,2) OR V(LIST:STAT)

(7*ABS(SGN('A'-2B'32')))(130 OR V(AB:A)))

The last formula is a number formula and may be substituted for a number (including in declarations).

CHAPTER 5. STATEMENTS

Statements are the mechanism in J73 which specifies the program algorithms. There are basically two types of statements, statements that compute values and statements that control program flow.

$$\begin{aligned} \underline{\text{statement}} &::= \left\{ \begin{array}{l} \underline{\text{simple statement}} \\ \underline{\text{named statement}} \\ \underline{\text{compound statement}} \end{array} \right\} \\ \underline{\text{simple statement}} &::= \left\{ \begin{array}{l} ; \\ \underline{\text{assignment statement}} \\ \underline{\text{goto statement}} \\ \underline{\text{return statement}} \\ \underline{\text{stop statement}} \\ \underline{\text{loop statement}} \\ \underline{\text{if statement}} \\ \underline{\text{switch statement}} \\ \underline{\text{procedure call statement}} \end{array} \right\} \end{aligned}$$

A statement consisting solely of a semicolon is a no operation statement. The remaining statements are described later in this chapter.

$$\begin{aligned} \underline{\text{named statement}} &::= \underline{\text{statement name}} : \underline{\text{statement}} \\ \underline{\text{statement name}} &::= \underline{\text{name}} \end{aligned}$$

Any statement may be named. A statement may have more than one name.

$$\underline{\text{compound statement}} :: \text{BEGIN} \left\{ \begin{array}{l} \underline{\text{statement}} \\ \underline{\text{declaration}} \\ \mu \end{array} \right\} \tilde{\beta} \left\{ \begin{array}{l} \{ \underline{\text{statement name}} : \} \tilde{\beta} \\ \mu \end{array} \right\} \text{END}$$

A compound statement groups a series of statements and/or declarations for treatment as a single statement. The END of a compound statement may also have a statement name and is treated as if a no operation statement followed the statement name.

ASSIGNMENT STATEMENT

An assignment statement is the method by which variables are assigned values.

$$\underline{\text{assignment statement}} ::= \underline{\text{variable}} \sim = \underline{\text{formula}} ;$$

An assignment statement indicates that the value of the formula is to be assigned to the variables. The formula is evaluated first, then any indices or base formulas required for the leftmost variable are computed, and then the formula value is assigned to the variable. Then the indices and base formulas for the second variable are computed and the formula value is assigned to the second variable. This sequence continues until all variables are assigned.

If the type of the formula does not correspond to the type of the variable, the compiler supplies the necessary conversions. Conversions are always applied to the original formula value. If the type of the variable is character or bit, the size of the formula must correspond. If the sizes do not match, the size is adjusted as described in Chapter 4.

If the variable is a bit or byte function variable and the number of bits (or bytes) parameter of the function variable is zero, no assignment is performed.

Given the following declarations

```
ITEM BOOL U 1;
ITEM UPPER S ==10;
DEFINE LIMIT "-50";
ITEM II U =4;
ITEM REAL F;
ITEM VAL S 5;
ITEM FORMAT C 30;
TABLE AA[10] S 24;
```



```

TABLE TABP[100] P 4; BEGIN
    ITEM CP C 12 N[0,0];
    ITEM FP F {0,3}; END
TABLE TABS[50]; BEGIN
    ITEM FS1 F;
    ITEM FS2 F;
    ITEM FS3 F; END

```

the assignments statements illustrated below yield results as explained in the comments:

```

BOOL=UPPER>LIMIT; "BOOL IS SET TO 1"
TABP[4]=TABS[I1+1]; "ENTRY 5 OF SERIAL TABLE TABS IS PADDED WITH A
                     WORD OF ZEROS ON THE LEFT AND ASSIGNED TO
                     PARALLEL TABLE TABP"
VAL.REAL =1.5; "1.5 IS CONVERTED TO INTEGER 1 FOR ASSIGNMENT TO
               VAL; 1.5 IS ASSIGNED TO REAL"
AA[I1],I1,AA[I1]=I1+1; "AFTER THIS STATEMENT, AA[4]=5, I1=5, AA[5]=6"
FORMAT='(X,316,F10.3,1,6H(COUNT=),A4)'; "THE CONSTANT IS PADDED WITH 1
                                           BLANK ON THE RIGHT AND ASSIGNED
                                           TO FORMAT"

```

GOTO STATEMENT

The goto statement is used to cause an unconditional transfer to the statement named.

goto statement ::= GOTO statement name ;

The results of a transfer to a statement name in an outer procedure will be described with procedure declarations. The following illustrates the goto statement.

GOTO LAB ;

STOP STATEMENT

The stop statement is used to terminate execution of a program.

stop statement ::= STOP ;

A stop statement may occur in either a program or a procedure and, depending on the system, executes a machine halt or a return to the operating system.

RETURN STATEMENT

A return statement provides the capability to exit from a procedure.

return statement ::= RETURN $\left\{ \begin{array}{c} \text{procedure name} \\ \mu \end{array} \right\}$;

procedure name ::= name

A return statement is permitted only within a procedure. Its effect is to terminate the execution of the procedure, set any actual output parameters from the formal parameters, and return control following the call.

If the return statement references a procedure name, the exit is from the procedure referenced. If a procedure name is not referenced, the innermost procedure is exited. The return statement is treated as if a goto statement had occurred referencing a statement name attached to the END associated with a compound procedure body.

A procedure name referenced in a return statement must be the name of a procedure in which the return statement is enclosed. If the return is from an outer procedure, any output parameters of the inner procedures containing the return statement are not set.

The return statement is undefined if the procedure being returned from is not active. Such a return can only occur if the procedure being exited was entered via an external definition for a statement name or inner procedure.

Examples of the return statement are

```
RETURN ;
RETURN SIN :
```

IF STATEMENT

The if statement provides for the conditional execution of a statement depending on a conditional formula.

$$\begin{aligned} \text{if statement} &::= \text{IF } \text{conditional formula} ; \text{conditional statement} \blacktriangleright \\ &\blacktriangleright \left\{ \text{ELSE } \frac{\text{else statement}}{\mu} \right\} \\ \text{conditional statement} &::= \text{statement} \\ \text{else statement} &::= \text{statement} \end{aligned}$$

If the conditional formula is true (the rightmost bit is 1), the conditional statement is executed. After execution of the conditional statement, execution continues with the statement following the if statement. Notice that for the true case, the else statement, if present, is skipped.

If the conditional formula is false (the rightmost bit is zero), the conditional statement is skipped; the else statement is, however, executed if one is present. In the event of an if statement within an if statement, the ELSE is associated with the inner if statement unless the inner statement is enclosed within BEGIN-ENDs.

The if statements below demonstrate nesting (indentation is used to illustrate nesting).

```

IF EMPLOYEE[NO]=V(MARRIED);
    IF NO'CHILD[NO]<>0;
        DEPENDENTS=NO'CHILD[NO]+2;
    ELSE DEPENDENTS=2;
ELSE IF EMPLOYEE[NO]=V(HEAD'HOUSE);
    GOTO HD'HSE;
ELSE DEPENDENTS=1;

IF FNC(AA,BB); BEGIN "TRUE IF LAST BIT OF FNC RESULT 1"
    IF AA+BB>=0;
        AA,BB=0; END
ELSE AA,BB=1; "IF NOT FNC(AA,BB)"

```

Notice in the second example that without the **BEGIN-END**, the **ELSE** would be associated with the inner **IF**.

SWITCH STATEMENT

The switch statement is the logical extension of the if statement. Instead of providing a two-way choice for statement conditionality as the if statement does, the switch statement selects one of several statements.

$$\begin{aligned} \text{switch statement} &::= \text{SWITCH } \underline{\text{numeric formula}} ; \left\{ \left\{ \frac{\text{statement name}}{\mu} : \right\} \tilde{\beta} \right\} \blacktriangleright \\ &\blacktriangleright \text{BEGIN } \left\{ \left\{ \left\{ \begin{array}{c} + \\ - \\ \mu \end{array} \right\} \frac{\text{integer}}{\mu} \right\} \left\{ \frac{\text{statement}}{\mu} \left\{ \begin{array}{c} \cdot \\ \mu \end{array} \right\} \right\} \tilde{\beta} \right\} \tilde{\beta} \blacktriangleright \\ &\blacktriangleright \left\{ \left\{ \frac{\text{statement name}}{\mu} : \right\} \tilde{\beta} \right\} \text{END} \end{aligned}$$

Each statement within the BEGIN-END is associated with an integer as a switch point. A bracketed integer denotes the switch point value for the statement following. Subsequent statements have a switch point value of successive integer

values. Every occurrence of a bracketed integer resets the switch point values; this resetting may result in a value greater or less than the previous statement but may not designate two statements to have the same switch point value. If the initial index is omitted, the first statement is associated with zero. Execution of the switch statement causes the numeric formula to be evaluated and its value used to select the related statement to be executed. After execution of the switch point statement, control simply falls into the next statement in source order (which is not necessarily in switch point order) unless a comma terminates the statement. Commas are used to cause control to continue after the switch statement as if an explicit goto statement had occurred which referenced a statement name on the switch statement END. For instance in the following switch, the "GOTO SWEND;" may be removed without changing the meaning of the switch statement.

```
SWITCH VALUE: BEGIN
    BEGIN AA=0; GOTO SWEND; END, "PT. 0"
    BEGIN AA=1; END, "PT. 1"
    BEGIN AA=4; GOTO L2; END, "PT. 2"
SWEND; END
```

Notice that the commas following the first and third switch point statements cause a transfer that can never be executed.

The result of a switch that selects an out of range switch point or a switch point within a gap created by a bracketed index is undefined.

A statement name may be attached to the switch statement BEGIN. When this statement name is referenced in a goto statement, the first statement following the BEGIN is executed. Statement names may also be applied to a switch point statement and referenced as any other statement name. The following switch might be used to process J73 item description abbreviations.

```
SWITCH LETTER: BEGIN
    ['S'] TYP=V(SIGN'INT);,
    ['U'] TYP=V(UNS'INT);,
    ['F'] TYP=V(FLOAT);,
    ['C'] TYP=V(CHAR); END
```

Notice that the switch statement above works regardless of the computer and the character encoding.

Another switch example is

```
LOOP: SWITCH WHICH(XX,YY);
    BEGIN
        "0"; "1"; "2"; "3" XX=0;,
        "4"; "5" YY=1; "6" GOTO L11;
    [10] "10" BEGIN XX=XX+1; GOTO LOOP; END
    L11: "11" YY=XX;
    [-7] "-7" IF YY<4; YY=YY+4;
        ELSE YY=4;,
        "-6";
    END
LOUT:
```

In the above switch, the points -6, -7, 0 to 6, 10 and 11 are valid; all other values for WHICH yield undefined results. For the values 0 to 3, XX is set to zero and the statement at LOUT is executed. The values 4 and -6 go to LOUT. Point 5 sets YY equal to 1 and falls into 6; point 6 transfers to the statement at L11. Point 10 increments XX and re-executes the switch. Point 11 sets YY to XX and falls into point -7. Point -7 executes the if statement and regardless of the truth of YY<4, eventually exits to LOUT.

LOOP STATEMENTS

J73 provides two forms of loop statements. The while loop is used to cause iterative processing of a statement as long as a condition is true. The for loop is used to provide an algorithm for assigning successive values to a loop control variable while iteratively executing a statement.

$$\text{loop statement} ::= \left\{ \begin{array}{l} \text{while clause} \\ \text{for clause} \end{array} \right\} \text{controlled statement}$$

$$\text{controlled statement} ::= \text{statement}$$

The controlled statement is the statement which is executed repetitively.

While Loop

The while loop provides a goto-less flow structure for the simple loop controlled by a condition.

while clause ::= WHILE conditional formula ;

The controlled statement is continuously executed as long as the conditional formula is true. The first two loops shown below are synonymous; either loop may be executed zero times since the test is made before every repetition. The third loop does not make a first time test so will always be executed at least once.

```

      WHILE LINK[AA]<>0; AA=LINK[AA];
TOP:  IF LINK[AA]<>0; BEGIN
      AA=LINK[AA]; GOTO TOP; END
LOOP: AA=LINK[AA];
      IF LINK[AA]<>0; GOTO LOOP;

```

For Loop

The for loop is a convenient and powerful control statement for a process which must be performed iteratively.

for clause ::= FOR control variable : control clause ;

control variable ::= item name

$$\text{control clause} ::= \left\{ \begin{array}{c} \text{initial phrase} \\ \mu \end{array} \right\} \left\{ \begin{array}{c} \left\{ \begin{array}{c} \text{increment phrase} \\ \text{replacement phrase} \end{array} \right\} \text{terminator phrase} \\ \left\{ \begin{array}{c} \text{terminator phrase} \\ \mu \end{array} \right\} \left\{ \begin{array}{c} \text{increment phrase} \\ \text{replacement phrase} \end{array} \right\} \end{array} \right\}$$

initial phrase ::= formula

increment phrase ::= BY numeric formula

replacement phrase ::= THEN formula

terminator phrase ::= WHILE conditional formula

The control variable of a for loop is a scalar item. The control clause provides the initial and successive values for the control variable and specifies the terminating condition. Execution of the loop continues until the condition indicates termination or until a transfer of control branches out of the loop (such as by a goto, return, or stop statement).

The loop control is effected as follows:

- the control variable is initialized by the initial phrase as if by an assignment statement.
- the terminator phrase is examined for loop termination.
- the loop controlled statement is executed.
- the increment/replacement phrase is used to modify the control variable.
- execution continues with the terminator phrase.

The control variable may be any type item. Its initial value in the loop is the value of the initial phrase formula, if present. Otherwise, the control variable value is its current value at entrance to the loop statement.

The formulas used in the increment, replacement, and terminator phrases are recomputed for each iteration of the loop.

An increment phrase denotes a value to be added to the control variable at each iteration; the value may of course be negative. The replacement phrase specifies a totally new value. Increment phrases are often used for counting or searching table entries. Replacement phrases are convenient for stepping along linked entries or structures.

The terminator phrase describes the condition for loop execution. Execution continues as long as the conditional formula is true. When false, control is transferred to the point following the loop statement. If the terminator phrase is omitted, execution continues until a transfer of control is encountered within the controlled statement.

The control variable may be used as an operand of any of the control clause formulas. The control variable may also be set or used as any other variable within the controlled statement.

The result of a branch into a loop statement is undefined. Procedures and functions may be called from a loop statement.

The meaning of the various combinations of control clauses is summarized in the chart below.

	<i>No terminator phrase</i>	<i>Terminator phrase</i>
No initial phrase No replacement phrase No increment phrase	1A. Leave control variable alone. Execute controlled statement "continuously".	1B. Same as 1A except test in accordance with terminator phrase before <u>every</u> execution of the controlled statement — even the first one.
Initial phrase only	2A. Initialize control variable. Execute controlled statement "continuously".	2B. Same as 2A except test as in 1B.
Replacement phrase only	3A. Leave control variable alone for the first execution. Before each subsequent execution of the controlled statement, <u>replace</u> the value of the control variable. Repeat executions "continuously".	3B. Same as 3A except test as in 1B.
Increment phrase only	4A. Same as 3A except <u>add</u> to value of control variable instead of <u>replacing</u> value.	4B. Same as 4A except test as in 1B.
Initial phrase and Replacement phrase	5A. Initialize control variable. Execute controlled statement once. Replace value of control variable before each subsequent execution. Repeat executions "continuously".	5B. Same as 5A except test as in 1B.
Initial phrase and Increment phrase	6A. Initialize control variable. Execute controlled statement once. Add to value of control variable before each subsequent execution. Repeat executions "continuously".	6B. Same as 6A except test as in 1B.

Following are some sample loop statements

```
FOR II:0 BY 1 WHILE II<=99;
  TAB[II]=0;
FOR NEXT:1 THEN LINK[NEXT] WHILE LINK[NEXT]<>0; ;
FOR DEG:45.0 BY -5.0 WHILE DEG>=-.5; ARRAY[DEG/5+.5]=SIN(DEG);
```

The first loop clears 100 entries of variable TAB. The second loop simply uses the loop control to locate the end of a linked list which is terminated by a link of zero. The third loop computes the SIN of degrees 0–45 in steps of 5 and assigns the value to ARRAY entries 0–9 (.5 is used to eliminate floating precision errors).

Other more complicated loops are shown below

```
FOR XX:0 BY 1 WHILE XX<=14;
BEGIN
  FOR YY:0 BY 1 WHILE YY<>XX;
```

```

BEGIN "TRANSPOSE MATRIX"
  TEMP=MATRIX[XX,YY];
  MATRIX[XX,YY]=MATRIX[YY,XX];
  MATRIX[YY,XX]=TEMP;
END
MATRIX[XX,XX]=0; "CLEAR DIAGONAL"
END

```

In the above loop, the inside loop controlled statement will be executed 105 times.

The following loop statement illustrates the Shell sort written in J73.

```

FOR DD: SORTNO/2 THEN DD/2 WHILE DD<>0;
BEGIN
  ZZ= SORTNO-DD-1;
  FOR II:0 BY 1 WHILE II<=ZZ;
  BEGIN
    JJ=II+DD;
    FOR KK:II BY -DD WHILE KK>=0;
    BEGIN
      IF TITLE[JJ]<=TITLE[KK]; GOTO TEST'I;
      CHAR=TITLE[JJ];
      TITLE[JJ]=TITLE[KK];
      TITLE[KK]=CHAR; JJ=KK; END
    TEST'I: END END
  END

```

PROCEDURE CALL STATEMENT

The procedure call statement is used to cause invocation of a procedure that is not a function.

$$\begin{aligned}
 \text{procedure call statement} &::= \text{procedure name} \left\{ \begin{array}{c} @ \text{ data base} \\ \mu \end{array} \right\} \rightarrow \\
 &\rightarrow \left\{ \begin{array}{c} (\text{actual input parameter } \sim) \\ (\left\{ \begin{array}{c} \text{actual input parameter} \\ \mu \end{array} \right\} : \text{actual output parameter } \sim) \\ \mu \end{array} \right\} ; \\
 \text{actual input parameter} &::= \left\{ \begin{array}{c} \text{statement name} \\ \text{procedure name} \\ \text{formula} \\ @ \text{ base formula} \\ \left\{ \begin{array}{c} \text{table name} \\ \text{block name} \end{array} \right\} \left\{ \begin{array}{c} @ \text{ base formula} \\ \mu \end{array} \right\} \end{array} \right\} \\
 \text{actual output parameter} &::= \text{variable} \\
 \text{data base} &::= \text{numeric formula} \\
 \text{base formula} &::= \text{numeric formula} \\
 \text{procedure name} &::= \text{name}
 \end{aligned}$$

Procedures are groups of statements which are callable from multiple points in a program. Procedures are constructed such that the data to be processed by the procedure statements is parameterized and defined at each call. These actual parameters must match the formal parameters of the called procedure in number, kind, and parameter list position. The matching of kind is described as follows.

- An actual parameter statement name must be passed corresponding to a formal parameter statement name. Statement name parameters are passed by name (address).
- Formal input parameters which are blocks or tables are also passed by name. The legal actual parameters corresponding to a formal parameter block or table must be a block or table name (if the block or table being passed is based, an explicit base address formula may be included), a named variable, a constant, or an @ followed by a formula whose value is to be passed as an address (in address units). The address which is passed for actual parameter blocks, tables, or named variables is the address which would result from the loc function applied to the parameter. Any base formulas are converted to integer if necessary. Caution should be used when passing an address of a variable which is dense packed or occupies multiple words in parallel table.

- If the formal input parameter is a procedure name, the actual parameter must be a procedure name with parameters which match those of the formal parameter procedure. Procedure names are passed by address.
- If the formal input parameter is an item name, the corresponding input parameter must be a formula. Parameters corresponding to item names are passed by value at invocation. Although the exact protocol of procedure invocation is not dictated by the language, the value of the actual parameter formula is computed, converted as necessary to match the formal parameter item, and assigned to the formal parameter item as if by an assignment statement.
- Formal output parameters may only be items; these parameters are passed by value at the procedure exit. The corresponding actual parameter must be a variable which is assigned the value associated with the formal parameter at the exit from the procedure. Any conversion required to match the parameter types is supplied by the compiler as for assignment statements.

If the procedure being called is itself a formal parameter procedure, the parameter matching and conversions of the actual parameters will be based upon the formal procedure's formal parameters.

The order of evaluating actual parameters is unspecified except that input parameters are computed before the call and output parameters (including indices, base formulas, and bit/byte function parameters) are computed after return from the call.

If the procedure being called does not exit normally (such as a goto an outer scope or formal parameter statement name), the actual output parameters are not assigned.

Procedures may be declared as based. This form of procedure (and only this form) must be called with a data base formula whose value is the address of the data base that the procedure is to use. These based procedures and corresponding based procedure calls are used to control reentrancy at the user level. This capability will be further described with procedure declarations.

Following are some examples of procedure calls:

```

RANDOM(:NUMBER); "RANDOM HAS NO INPUT PARAMETERS"
IO('FILE,XYZ',V(READ),DATA[II],WORDSIZE(DATA):STATE);
    "IO (FILE,ACTION,ADDRESS,NO'WORDS:IO'STATUS)"
SORT@SPACE(TAB,TABX); "THE VALUE OF SPACE IS USED BY SORT AS
    THE ADDRESS OF ITS DATA"
DUMP(@LOC(FLIGHT[II+1]),3B'1000'); "DUMP (LOCATION,SIZE); LOCATION
    MUST BE NAME PARAMETER"
DATE(:MONTH,DAY,YEAR); "DATE HAS 3 OUTPUT PARAMETERS"
WAIT; "PARAMETERLESS PROCEDURE CALL"

```

CHAPTER 6. PROGRAM, PROCEDURE DECLARATIONS

A J73 compilation may be a program that is invoked by an operating system or a procedure called from a program or other procedure.

$$\begin{aligned} \text{JOVIAL compilation} &::= \left\{ \begin{array}{l} \text{compool} \\ \left\{ \frac{\text{directive}}{\mu} \right\} \tilde{\beta} \left\{ \begin{array}{l} \text{program} \\ \text{procedure declaration} \end{array} \right\} \end{array} \right\} \\ \text{program} &::= \text{PROGRAM } \text{program name} ; \left\{ \begin{array}{l} \text{statement} \\ \text{declaration} \end{array} \right\} \\ \text{program name} &::= \text{name} \end{aligned}$$

Within a compilation (program or procedure), internal procedure declarations may be nested to any level. Procedures must be declared before they are referenced. Within a program, a stop statement is used to terminate a program or procedure and return to the system. A stop statement is inserted by the compiler at the end of the final statement of a program. A return statement may not be used within a program but may be used within any procedure.

NAME SCOPE

Before entering into the description of procedure declarations it is necessary to discuss the concept of name scope. Each program or procedure establishes a scope or region in which a name has meaning. Any name declared within a program or procedure has a meaning for the entire program or procedure. Procedures may be declared within a program or procedure and in so doing create a new scope of names. Names declared in this inner scope are unknown to the outer scope. However, if a name is not redeclared in an inner scope, an outer scope name retains its outer scope definition within the inner scope. As procedures are nested within other procedures, a hierarchy of name scopes is developed. When a name occurs in a context other than declaration, its attributes are determined by the innermost preceding declaration for the same name in a containing scope. Declarations which occur within an inner scope or disjoint scope do not satisfy a name reference.

In addition to the scope levels within a compilation, names may be predefined in a compool scope. The compool scope encloses the program (or procedure) and contains any names extracted from the compool. Compools are described in the next chapter.

Names may not be multiply defined in a scope.

PROCEDURE DECLARATION

A procedure declaration serves to declare a processing algorithm that may be invoked from multiple places. Procedures are compiled as closed subroutines which are executed and then normally return to the caller. The code of a procedure is not duplicated at the point of a reference. A procedure may only be invoked by a procedure call statement. A procedure cannot be executed as a result of sequential statement execution; the compiler will supply any necessary branches around a procedure.

$$\begin{aligned} \text{procedure declaration} &::= \text{procedure clause } \text{procedure body} \\ \text{procedure clause} &::= \text{PROC } \text{procedure name} \left\{ \frac{\text{data allocator}}{\mu} \right\} \rightarrow \\ &\rightarrow \left\{ \begin{array}{l} \left(\left\{ \frac{\text{formal input parameter}}{\mu} \right\} : \text{formal output parameter } \sim \right) \\ \left(\text{formal input parameter } \sim \right) \left\{ \begin{array}{l} \text{item description} \\ \mu \end{array} \right\} \left\{ \begin{array}{l} = \\ \left\{ \begin{array}{l} + \\ - \\ \mu \end{array} \right\} \text{constant} \end{array} \right\} \mu \end{array} \right\} \end{array} \end{aligned}$$

$$\begin{aligned}
 \text{procedure name} &::= \text{name} \\
 \text{data allocator} &::= \left\{ \begin{array}{l} \text{IN} \\ \text{RESERVE} \\ @ \\ \mu \end{array} \right\} \\
 \text{formal input parameter} &::= \left\{ \begin{array}{l} \text{data name} \\ \text{statement name} \\ \text{procedure name} \end{array} \right\} \\
 \text{formal output parameter} &::= \text{item name} \\
 \text{data name} &::= \left\{ \begin{array}{l} \text{item name} \\ \text{table name} \\ \text{block name} \end{array} \right\} \\
 \text{procedure body} &::= \left\{ \begin{array}{l} \text{statement} \\ \text{declaration} \end{array} \right\}
 \end{aligned}$$

The procedure clause establishes the name of the procedure, describes the allocation of its data, and defines the number and position of its formal parameters. If the procedure is to be a function, an item description appears in the clause to describe an internal item which upon exit will yield the function result. The procedure body contains the declarations of the procedure's formal parameters, its local data, and the statements which perform the procedure algorithms.

As noted in the syntax, a procedure clause for a function does not allow output parameters; the function result is the sole output of a function. In the following description, unless stated otherwise, procedures and functions are treated synonymously.

Each procedure establishes a new scope of names, i.e., within a procedure a name may be declared which duplicates and thereby overrides an outer scope name. Procedures may be nested to any level.

Procedure Data

Unless otherwise specified, procedure declared data is allocated as reserve data (including the function result item) and may be preset. This presetting occurs just once and not at every entrance. Any data within a procedure may be declared individually as based or as IN the current procedure. This local data cannot be assumed to retain values after a procedure exit.

In addition to the programmer declared data, there exists for every procedure a set of data which comprises such compiler generated data as temporary cells, dynamic parameter lists, register save areas, name parameter addresses, etc. This compiler generated data may be considered as being in an unnamed block which by default is allocated as local data.

J73 provides further control over both this unnamed data as well as the declaration data. The data allocator in a procedure heading is used to specify the allocation of all procedure data. A data allocator of RESERVE has no effect on procedure data since reserve data is the default. This default may be changed by using an IN as the data allocator indicating that all unnamed data as well as declared data which has no individual allocation specifier is to be allocated as local data. Although the language does not force dynamic space access, or even static space sharing, of this local space between procedures, programmers are cautioned not to take advantage of compilers which allocate local space as RESERVE space since the next compiler might choose to handle this local data space differently. A specification of IN is used primarily to inform the compiler that some form of space sharing may be, but need not be, applied to minimize data space requirements.

Based Procedures

By a similar mechanism, procedure data space may be described as based. The @ used as the data allocator in a procedure heading indicates that the unnamed data space, the local data (declared as IN), and the unspecified data is to be based. The location to be used for this space is passed to the based procedure by a based call indicated by inserting "@ formula" immediately following the procedure name. The value of this formula will be used to address all of the procedure's local space.

The placement of an @ in a procedure clause has further impact on procedures declared within this based procedure. All inner procedure local space is allocated as part of the containing based procedures local space. DSIZE of a based procedure (DSIZE may only be applied to based procedures) is the total of the following space:

- All compiler generated data for the based procedure and for any contained procedure.
- The IN data space for the based procedure and for any contained procedure.
- The default declared data space for the based procedure.

A based procedure may not be declared within a based procedure unless the inner procedure is a formal parameter or external reference.

Based procedures are used to minimize dynamic data space requirements and to control reentrancy at the user level. This is performed by passing the space required by the based procedure at its call. This space might be acquired dynamically from the system or by simply allocating a large table for use as available space. Management of this space to minimize a program's requirements or to achieve user level reentrancy is the responsibility of the user.

The following example illustrates the allocation and attributes of program data. In this program A1, B1, C1, D1, E1, and A3 are in reserve. A2, B2, C2, D2, and E2 are based and have no allocated space. B3 and the compiler generated space for BB may overlay C3 and CC's generated space if execution of BB does not result in a call on CC or vice versa. D3, D4, DD's generated space, E3, and EE's generated space are in an unnamed data block whose location is passed as the call to DD; DSIZE of DD is the total length of this unnamed block.

```

PROGRAM AA;
BEGIN
  ITEM A1 U;
  ITEM A2 @ A1 U;
  ITEM A3 IN U;

  PROC BB;
  BEGIN
    ITEM B1 U;
    ITEM B2 @ B1 U;
    ITEM B3 IN U;
  END

  PROC CC IN;
  BEGIN
    ITEM C1 RESERVE U;
    ITEM C2 @ C1 U;
    ITEM C3 U;
  END

  PROC DD @;
  BEGIN
    ITEM D1 RESERVE U;
    ITEM D2 @ D1 U;
    ITEM D3 U;
    ITEM D4 IN U;

    PROC EE;
    BEGIN
      ITEM E1 U;
      ITEM E2 @ E1 U;
      ITEM E3 IN U;
    END
  END
END
END

```

Procedure Parameters, Function Results

J73 provides an extensive procedure parameter capability. The attributes of a procedure's formal parameters are specified by a declaration within the procedure body. All formal parameters except those which are statement names must be declared prior to their first reference. These declarations (explicitly or by default) establish the mechanism used for parameter passing.

Blocks, tables, statement names, and procedures are passed as call by name at call. This form of parameter passing is performed by evaluating the name (address) of the actual parameter at the point of the call and passing this address to be used as the address of the formal parameter.

Item parameters are classified as call by value parameters. A call by value parameter may be either an input parameter or an output parameter. The formal input parameters are assigned the value of the actual parameter at invocation; the value of the formal output parameter item is assigned to the actual parameter item at the exit from the procedure. Item parameters, and only item parameters, may appear both as an input parameter and as an output parameter; in no other instances may a name appear twice in a formal parameter list.

Actual parameters must match in position, number, and kind as detailed below with the called procedure's formal parameters.

A statement name must be passed opposite a formal parameter statement name.

A table or a block declared as a formal input parameter may not have an allocation specifier and may not be preset or overlaid. A formal parameter table or block is treated similar to a based table or block except that the address is established at the procedure invocation by the corresponding actual parameter and may not be overridden with an explicit base formula. The corresponding actual parameter must be a block or table name (with an optional base formula), a named variable, a constant, or an @ followed by a base formula whose value is to be passed as an address. Blocks, tables, variables, and constants are passed as if the loc function had been applied and the result is passed as an address. During the computation of these addresses, base formulas and indices are converted to integer if necessary. Within the procedure body the data structure and attributes of the formal parameters are treated as declared; only the address of the actual parameter is utilized. The user must beware of formal named parameters whose attributes do not match the corresponding actual parameter; particularly in the instance of named variables which are dense packed or are contained in the non-contiguous multiple words such as in parallel tables.

Item parameters are declared as any other local item except they may not be declared as based unless an implicit base is also included. These parameters may be overlaid and, if reserve data, may be externally defined and/or preset with an initial value. Notice that a preset value in an item description of a formal input parameter has little utility unless the item overlays some outer scope data or is externally defined. If the item is a formal input parameter, its value will be assigned the value of the corresponding actual parameter at the procedure invocation. The actual parameter must be a formula; any conversion or size adjustment required will be supplied as for an assignment statement. If an item is a formal output parameter, its value at the return from a procedure will be assigned to the corresponding actual output parameter variable. Any conversions or size adjustments are implied as for an assignment statement. An item may be both a formal input and a formal output parameter.

The actual parameter passed corresponding to a formal input parameter procedure name must be a procedure. A formal parameter procedure is declared as a nested skeleton procedure declaration; it may not contain any statements in its procedure body and may not be externally defined. All of its parameters which are not statement names must be declared in the formal parameter procedure's body. The actual parameter procedure's formal parameters must match the formal procedure's formal parameter. At a call to a formal parameter procedure, the associated actual parameters are matched with the formal parameter procedure's formal parameters. A formal parameter procedure call is treated as if the corresponding actual parameter procedure was called directly. The example which follows should clarify the above.

```
PROC ABLE(BAKER); BEGIN
  PROC BAKER(Charley:DOG);
    ITEM DOG F;
    ITEM FOX S;
  L1: BAKER(L1:FOX); END
```

Procedure ABLE must be called with a procedure which has one input parameter — a statement name — and a single floating output parameter. Matching of parameters for the call on BAKER at L1 will be performed using the description of BAKER's formal parameters.

A function result is a special form of value output parameter. The existence of an item description in the procedure clause denotes the procedure to be a function and effects the declaration of a local item whose name is the same as the procedure name. This function item has all the characteristics of any other local item with no allocation specifier. The value of this function item at the return from a function is the result of the function call. A function value input parameter may bear the same name as the function and refers to the function result item.

Procedure Body

The statement(s) of a procedure body are executed when a procedure is invoked. At completion of this execution, in absence of a return statement, a return statement is assumed. Any formal output parameters are assigned to the actual output parameters and execution continues at the statement following the call.

If a goto statement references an outer scope statement name or a parameter statement name, the output parameters of this procedure or any procedures in the procedure call order which are skipped in transferring to the indicated statement are deactivated and the setting of any output parameters is bypassed. A return statement which references an outer procedure also causes the bypassing of output parameters for the inner procedures being exited.

Deactivation of a procedure occurs when a procedure is exited normally by a return or when a goto statement references an outer scope or parameter label. Any local procedure data must be considered lost and any local values gone.

Some sample procedures are shown on the following page.


```

PROC TRIG(DEG:SIN,COS); BEGIN
  DEFINE PI "3.14159265";
  ITEM JJ U;
  ITEM DEG U;
  ITEM RAD F;
  ITEM FACTR F;
  ITEM SIN F;
  ITEM COS F;
  "THIS PROCEDURE COMPUTES BOTH SIN AND
  COS OF ANGLE DEG USING A TAYLOR SERIES"
  SIN=0.0; COS=1E0;
  RAD=DEG*PI/1.8E2;
  FACTR=1.0;
  FOR JJ:1 BY 1 WHILE JJ<=20; BEGIN
    FACTR=RAD*FACTR/JJ;
    IF JJ; BEGIN
      SIN=SIN+FACTR;
      FACTR=-FACTR; END
    ELSE COS=COS+FACTR; END END

PROC ACKER(MM,NN) U; BEGIN
  "THIS FUNCTION COMPUTES ACKERMAN'S FUNCTION
  AS DEFINED BY:
    IF M=0,A(M,N)=N+1
    ELSE IF N=0,A(M,N)=A(M-1,1)
    ELSE A(M,N)=A(M-1,A(M,N-1))."
  ITEM MM U;
  ITEM NN U;
  TABLE VALUE[9] U;
  TABLE PLACE[9] S;
  OVERLAY VALUE:ACKER; "VALUE[0] OVERLAYS THE FUNCTION RESULT"
  IF MM=0; VALUE[0]=NN+1;
  ELSE BEGIN
    VALUE[0]=1;
    PLACE[0]=0;
  LOOP:  VALUE[0]=VALUE[0]+1;
        PLACE[0]=PLACE[0]+1;
        FOR II=0 BY 1 WHILE II<=MM-1; BEGIN
          IF PLACE[II]=1; BEGIN
            VALUE[II+1]=VALUE[II];
            PLACE[II+1]=0;
            IF II=MM+1; GOTO CHECK;
            GOTO LOOP; END
          IF PLACE[II]<>VALUE[II+1]; GOTO LOOP;
          VALUE[II+1]=VALUE[II];
          PLACE[II+1]=PLACE[II+1]+1; END "II"
        CHECK: IF NN<>PLACE[MM]; GOTO LOOP; END END

```

DEFINE DECLARATION

J73 provides a source macro capability which allows for creating and modifying source by the substitution of a parameterized source string.

```

define declaration ::= DEFINE define name ➤
➤ { ( formal define parameter ~ ) } "define string" ;
      μ
define name ::= name
formal define parameter ::= letter
define string ::= character ~

```

Any reference to the define name after the semicolon of the define declaration is a define call and the define string is substituted in its place.

A comment is not permitted in a define declaration from the point of the define name until after the define string.

A define may have single letter parameters; their reference within a define string is indicated by a single exclamation point preceding the letter parameter. The existence of an exclamation point or a quotation mark within a define string must be indicated by two exclamation points or two quotation marks, respectively.

Define names obey the scope rules. A define call will not be recognized where the name being declared is expected. Within a procedure heading, define calls are not permitted where formal parameter names are expected in the parameter list. Define calls are also not recognized within a comment, a character constant, or as part of a name, key word, or status.

$$\begin{aligned} \text{define call} &::= \text{define name} \left\{ \left(\left\{ \frac{\text{actual define parameter}}{\mu} \right\} \sim \right) \right\} \\ \text{actual define parameter} &::= \left\{ \begin{array}{l} \text{character} \sim \\ \text{"character"} \sim \end{array} \right\} \end{aligned}$$

A define call causes the substitution of the define string in place of the call. If the define has parameters, the actual define parameters are substituted in place of the formal parameters.

If a define has formal parameters, the define call must include a parentheses enclosed actual parameter list. Individual actual parameters may be omitted indicating that no characters are to be substituted; the separating commas must be present in a define call.

If the actual parameter is enclosed in quotation marks, all characters enclosed are substituted for the formal parameter. If the actual parameter is not within quotation marks, the parameter consists of those characters beginning with the first non-blank character and ending at, but not including, the first comma or right parenthesis. Exclamation points are treated as any other character within an actual define parameter; however, a quotation mark must be doubled within a quotation mark enclosed actual parameter.

Symbols may not be created by the juxtaposition of the ends of the define string and the context of the define call. No such restriction exists for parameter substitution.

Some sample define declarations and define calls are shown below.

```
DEFINE P2 "4";
DEFINE P1 "3.14159265";
DEFINE SIGN(A) "BIT(!A,0)";
DEFINE LIKE(X,L,U,P)
  "TABLE TAB!X[L!U] !P;
  BEGIN
    ITEM ITM!X F=P!;
    ITEM CHR!X C 4;
  END";
SIGN(AA[INDX])=0;
LIKE(0, "(P2*2)",P)
LIKE(A, "V(LIST:A):", "V(LIST:Z)");
```

The source resulting from the three define calls after expansion is shown below.

```
BIT(AA[INDX],0)=0;
TABLE TAB0[(4*2)] P;
BEGIN
  ITEM ITM0 F=3.14159265;
  ITEM CHR0 C 4;
END
TABLE TABA[V(LIST:A):V(LIST:2)] ;
BEGIN
  ITEM ITMA F=3.14159265;
  ITEM CHRA C 4;
END
```

Define strings and define parameters may reference other define names but circular defines are illegal.

NAME DECLARATION

The name declaration is used to protect statement names from conflict with outer scope names.

name declaration ::= NAME statement name \sim ;

Each statement name which appears in the name declaration is established as a current scope statement name. The name declaration must be used if a statement name duplicates an outer scope name.

In the following program, the goto statement would have branched to the statement name ABC if the name declaration had been omitted.

```

PROGRAM PRG;
BEGIN
  DEFINE AA "ABC";
  PROC PRC;
  BEGIN
    NAME AA;
    GOTO AA;
    :
    :
    :
  AA:
    :
    :
    :
  END
ABC:
END

```

EXTERNAL DECLARATIONS

External declarations are used to make names from one program accessible to other programs.

external declaration ::= \blacktriangleright

$$\blacktriangleright \left\{ \begin{array}{l} \text{DEF} \\ \text{REF} \end{array} \right\} \left\{ \begin{array}{l} ; \\ \text{data declaration} \\ \text{name declaration} \\ \text{procedure declaration} \\ \text{define declaration} \\ \text{status list declaration} \\ \text{BEGIN} \left\{ \begin{array}{l} ; \\ \text{data declaration} \\ \text{name declaration} \\ \text{procedure declaration} \\ \text{define declaration} \\ \text{status list declaration} \\ \mu \end{array} \right\} \tilde{\beta} \text{ END} \end{array} \right\}$$

A DEF declaration is used to make the address of a name in a program available for reference by another program. DEF may not be applied to local or based data. A REF declaration is used to declare entities whose address is located in an independent compilation. No space is allocated for REF data. A REF declaration may not be declared within a REF declaration.

No special meaning is ascribed to status list and define declarations placed in an external declaration.

An implied DEF is assumed for the outermost scope name (program or procedure).

Caution should be taken when using a DEF declaration for procedure and statement names. If entrance to a procedure is through use of a DEF statement name, the procedure has not been properly invoked and any attempt to reference local data or execute a normal procedure return may yield undefined results. A similar situation may occur if a program is entered via a DEF procedure containing an out of scope goto.

The procedure body of a REF procedure declaration contains the declarations of its formal parameters; the procedure body must not contain any statements.

External declarations are illustrated below.

```
DEF BLOCK COM;
  BEGIN
    ITEM FUEL U;
    TABLE ROOTS[10]; BEGIN
      ITEM POS'ROOT F;
      ITEM NEG'ROOT F;
      ITEM IMAG'ROOT F; END
    DEF ITEM DATA C 4;
  END
DEF PROC FUNC(AA,BB) U;
  BEGIN
    REF PROC SIN(DD) F; ITEM DD F;
    PROC AA(:EE); ITEM EE U; "PARAMETER PROC"
    ITEM BB U;
    IF BB<0; AA(:FUNC);
    ELSE FUNC=SIN(BB);
  END
```

In the first example, the name and address of both COM and DATA are made external. In the second example the name and address of FUNC is made external; the ref procedure declaration for SIN would allow interfacing with an independently compiled function whose name was SIN.

CHAPTER 7. COMPOOL

A compool is a facility nearly unique to JOVIAL which allows for the creation of one or more preprocessed common data base descriptions. The compool processor accepts primarily standard J73 declarations. The compool source contains two kinds of information. First data declarations which are common to two or more programs may be described in a compool. In addition, any external procedures or functions may be declared in the compool. The compool process involves essentially a compilation of the compool source. The compool processor creates two forms of output. One output of this processor is an object module (relocatable file) containing space reservation for the data declared in the compool and any presets defined for this compool data. The second output is a special file containing names declared in the compool and their attributes for use by the compiler during subsequent compilations which refer to the names declared in the compool. This special file is designed to conform to the compiler's internal data base and is ordered to minimize the access time required to process the compool information.

COMPOOL DIRECTIVE

The compool directive is used to denote the compool files to be used and the names of compool entities whose attributes are desired.

$$\begin{array}{l} \text{compool directive} ::= \text{! COMPOOL} \left\{ \begin{array}{l} \left\{ \frac{\text{compool file}}{\mu} \right\} \left\{ \frac{\text{name}}{(\text{name})} \right\} \sim \\ \left(\frac{\text{compool file}}{\mu} \right) \end{array} \right\} ; \\ \text{compool file} ::= \text{character constant} \end{array}$$

The compool file is described by a character constant which names the file in whatever format is native to the compiler host system. The character constant might contain information such as file name, file version, file type, library or catalog name, etc. The format of this file description will be described in Appendix C — System Dependencies.

If the compool file specifier is absent from the directive, there must be a default system dependent compool file.

If the compool file specifier is enclosed in parentheses (or the compool file and name list are omitted), all names in the named compool (or default compool) and their associated attributes will be obtained for use by the program being compiled. Programmers are cautioned that this specification may bring in more names and attributes than are really required and cause unnecessary compiler space utilization.

The list of comma separated names indicate those names whose attributes are required from the compool. The names in the list may be of status lists, defines, items, tables, table items, blocks, statements (from a name declaration), and procedures. Implied by any name reference is the acquisition of any additional name and attribute that is required for the proper processing of the indicated name. For instance, if a table item name is indicated, it is necessary to also obtain the table description for structure and dimensioning attributes. Associated status lists must be accessed. Data contained in blocks require block description. Procedures names imply parameter and any function result attributes. These implied names and descriptions will be supplied automatically.

The only names and attributes which are not automatically obtained but are required are those for a compool contained name which occurs in a compool define string. Since the meaning of a definition is not known until the actual define call, the compiler cannot supply any related and/or required names. Caution should be taken in the use of compool defines for this reason.

Any names in the list that are enclosed in parentheses denote that all names and attributes within this structure are required. An enclosed block name signifies that all declarations contained within this block should be brought in, i.e., contained blocks, tables, items, defines, status lists. An enclosed table also denotes any contained table items, defines, and status lists. No significance is associated to the parentheses enclosure of names of items, statements, procedures, defines, or status lists.

As described in the previous chapter, names from the compool are entered into a scope created outside the outermost scope of a compilation. This permits any compool declared name to be overridden by declarations within the compilation source.

COMPOOL SOURCE

The source used for creating a compool is a proper subset of J73 source except that a compool clause may not appear in the source for a compilation.

$$\begin{aligned}
 \text{compool} &::= \text{compool clause} \left\{ \begin{array}{l} \text{compool declaration} ; \\ \text{BEGIN} \left\{ \frac{\text{compool declaration}}{\mu} \tilde{\beta} \right\} \text{END} \end{array} \right\} \\
 \text{compool clause} &::= \text{COMPOOL compool name} ; \\
 \text{compool declaration} &::= \left\{ \begin{array}{l} \text{overlay declaration} \\ \text{data declaration} \\ \text{name declaration} \\ \text{procedure declaration} \\ \text{define declaration} \\ \text{status list declaration} \\ \text{def declaration} \end{array} \right\} \\
 \text{compool name} &::= \text{name} \\
 \text{def declaration} &::= \text{DEF} \left\{ \begin{array}{l} \text{def declaration} \\ \text{BEGIN} \left\{ \frac{\begin{array}{l} \text{data declaration} \\ \text{define declaration} \\ \text{status declaration} \\ \text{def declaration} \end{array}}{\mu} \tilde{\beta} \text{END} \end{array} \right\}
 \end{aligned}$$

A compool declares its name and those names common to multiple programs that are to be contained and described in the compool. The allocation specifier in a compool data declaration may not indicate IN or RESERVE. All compool data which is not declared based or is not a procedure parameter will be reserve data and may be preset.

Compool data which is not based is treated as if it were declared in a block whose name is the compool name.

A compool procedure declaration obeys the same conventions as a ref or parameter procedure declaration (dynamic statements are not permitted). All parameters except statement names must be declared.

An example compool is shown below.

```

COMPOOL CPL1;
BEGIN
  TABLE TAB1[1:5] P; BEGIN
    ITEM I1 S;
    ITEM C4 C 4='A','B'; END
  ITEM TABX U=5;
  OVERLAY TABX,TAB1;
  STATUS LIST V(A), V(B), V(C);
  DEF TABLE LETTER[25] U 3 LIST;
  TABLE BASED@ [9] P 2;
  BEGIN
    ITEM BAS'ITM C 10 [0];
    ITEM CHAR1 C 1 [0];
  END
  PROC SIN(ALPHA) F; ITEM ALPHA F;
END

```

In the above example, the allocation of TABX and TAB1 are fixed with respect to each other because of the overlay declaration. LETTER, TAB1, and TABX are allocated to the block CPL1. The allocation of LETTER with respect to TABX and TAB1 is unspecified and only known to be within block CPL1. Table BASED is based and hence, not allocated in block CPL1.

The J73 compilers will permit multiple compool directives but each must denote a different compool file. It is illegal to include the same name from two compools either by explicit reference in the compool directive name list or implicitly by the automatic inclusion to completely satisfy an explicit reference or a parentheses enclosed name.

For instance, given the compool and directives below, the second directive causes three illegal names because of name duplication.

```

COMPOOL A1; BEGIN
  STATUS S2 V(A), V(B), V(C);
  ITEM I3 S 3 S2;
  TABLE T4[5]; ITEM I5 F;
  PROC P1; ; END

COMPOOL B1; BEGIN
  ITEM S2 F;
  BLOCK B2; ITEM T4 C;
  NAME P1; END

!COMPOOL 'A1' I3,I5,P1;
!COMPOOL 'B1' S2,(B2),P1;

```

The duplication is caused as follows:

- S2 was brought in from A1 with I3 and from B1 by explicit reference.
- T4 is brought from A1 by I5 and from B1 by enclosing B2 in parentheses.
- P1 is brought in explicitly by both directives.

COMPOOL OUTPUT

The processing of a compool results in the generation of two forms of output. The first and most obvious form is an object module which represents the data space associated with the compool block. Any presets in the compool are included in this object module. The compool object module will contain an external definition for the compool block name and any data name declared as externally defined (using the def declaration).

The second output of the compool process is the file to be used by the compiler for compool name resolution. Any reserve data entered from the compool will be treated as data occurring in a REF block declaration whose name is that of the compool. Based compool data, status lists, and defines will simply be treated as if declared in a program except that as with all compool names, the names will be in compool scope and may be overridden by a program declaration for the same name. Compool statement names (declared via the name declaration) and procedures will be entered as REF names and procedures.

CHAPTER 8. DIRECTIVES

Directives are used to provide supplemental information to a compiler about the program being compiled. Directives are also used to provide compiler control. The general form for a directive is:

directive ::= ! directive key symbol ~ ;

The directive keys processed in J73/1 are described below.

directive key ::= {

COMPOOL
COPY
SKIP
BEGIN
END
LIST
UNLIST
EJECT
LINKAGE
TRACE

}

Directives may only occur where a statement or declaration is legal and not within a simple statement or declaration. Directives are treated as comments as far as the language system is concerned.

The compool directive is described in Chapter 7. The remaining directives are described below.

SOURCE DIRECTIVES

Several directives are provided in J73 to control the processing of source. These directives accommodate accessing of an auxiliary source file for merging into the program and control conditional compilation.

Copy Directive

The copy directive enables the independent maintenance of source that is common to more than one program.

copy directive ::= ! COPY copy file ;

copy file ::= character constant ;

The character constant comprises the system dependent description of the file to be copied. The copy directive might be considered as a define call; the copy directive is expanded at the point of its occurrence by substituting the entirety of the file being copied.

Skip, Begin, End Directives

The skip directive is used to control the conditionality of source processing. The begin and end directives are used to enclose conditional source.

skip directive ::= ! SKIP { $\frac{\text{letter}}{\mu}$ } ;

begin directive ::= ! BEGIN { $\frac{\text{letter}}{\mu}$ } ;

end directive ::= ! END ;

The skip directive provides 27 levels of source conditionality. This directive works in conjunction with the begin and end directives to cause enclosed source to be ignored. A skip directive with a letter will suppress the processing of all source following a begin directive containing the same letter up to the matching end directive. A skip directive with no letter activates all begin directives; a begin directive with no letter is activated only by a skip directive with no letter.

Begin-end directive pairs may be nested. An end directive is associated with the most recent preceding begin directive. Within an active begin-end directive set, enclosed begin directives are recognized only for the purpose of matching end directives.

Since define calls are not recognized within an active begin-end pair, any actual define parameters which are not comprised of symbols must be enclosed in quote-marks within active begin-end directives.

The example below illustrates the use of these conditional source directives.

```

!SKIP U; "SUPPRESS 1108 DEFINES"
!BEGIN U; "UNIVAC 1108 DEFINES"
    DEFINE CW "4"; "CHARS/WORD"
    DEFINE BC "9"; "BITS/CHAR"
    DEFINE BW "36"; "BITS/WORD"
!END;
!BEGIN D; "DEC-10 DEFINES"
    DEFINE CW "5"; "CHARS/WORD"
    DEFINE BC "7"; "BITS/CHAR"
    DEFINE BW "36"; "BITS/WORD"
!END;
TABLE CONV[255] ((25+CW-1)/CW);
ITEM MSG C 25 [0];

```

The source above could be used to specify machine independent declarations controlled by a set of defines. The "!"SKIP U;" suppresses the "U" set of defines; a skip directive using D would suppress processing of the "D" set of defines.

LISTING DIRECTIVES

Several directives provide control over the listing of program source.

```

eject directive ::= ! EJECT ;
nolist directive ::= ! NOLIST ;
list directive ::= ! LIST ;

```

The eject directive causes a page eject on the source listing before the next source line. This directive is ignored if the source listing is being suppressed.

The nolist directive causes the suppression of the source listing beginning with the next source line.

The list directive causes the listing of source to begin with the next source line.

The following set of directives would suppress the listing of copied source.

```

!NOLIST; !COPY 'ABC.JOV';
!LIST;

```

Any eject directives in file ABC.JOV would be ignored.

LINKAGE DIRECTIVE

The linkage directive is used to indicate a procedure that does not obey J73 linkage conventions.

```

linkage directive ::= !LINKAGE symbol~ ;

```

The character string denotes the system dependent linkage type to be used for this procedure. The linkage directive is only permitted for external (REF) procedures, compool procedures, and formal parameter procedures. The linkage directive must be placed between a procedure clause and the procedure body and is effective for only that procedure.

An example of the use of the linkage directive is:

```

REF PROC SIN(DEG) F;
    !LINKAGE FORTRAN;
    BLOCK DEG; ITEM DEGREES;

```


TRACE DIRECTIVE

The trace directive provides a run-time facility to trace program execution and monitor data assignment.

$$\text{trace directive} ::= ! \text{ TRACE } \left\{ \underset{\mu}{(\text{conditional formula})} \right\} \text{ name } \sim ;$$

The conditional formula is examined dynamically to control the listing of trace information.

If a name in the list applies to a statement, the execution of the indicated statement is noted on a system dependent listing whether the statement was branched to or fallen into.

Each call to a procedure that is named in the list will be similarly noted.

If the name in the trace list is a data name, modification of the datum will be noted along with its new value. If the name is a table name, modifications of both the table and table items will be noted. For block names, modification of any enclosed data will be noted. The modifications may occur as the result of an assignment or by use as an actual output parameter.

The description of the trace output will be described in Appendix C.

The trace directive is illustrated below:

```

ITEM LOOP U;
DEFINE VECMAX "999";
TABLE VECTOR[VECMAX];
BEGIN
    ITEM LEFT F;
    ITEM RIGHT F;
END
!TRACE (LOOP AND LOOP<100) VECTOR;
FOR LOOP:0 BY 1 WHILE LOOP<=VECMAX;
BEGIN
    LEFT[LOOP]=AA*BB+CC;
    IF AA=0; RIGHT[LOOP]=0;
END

```

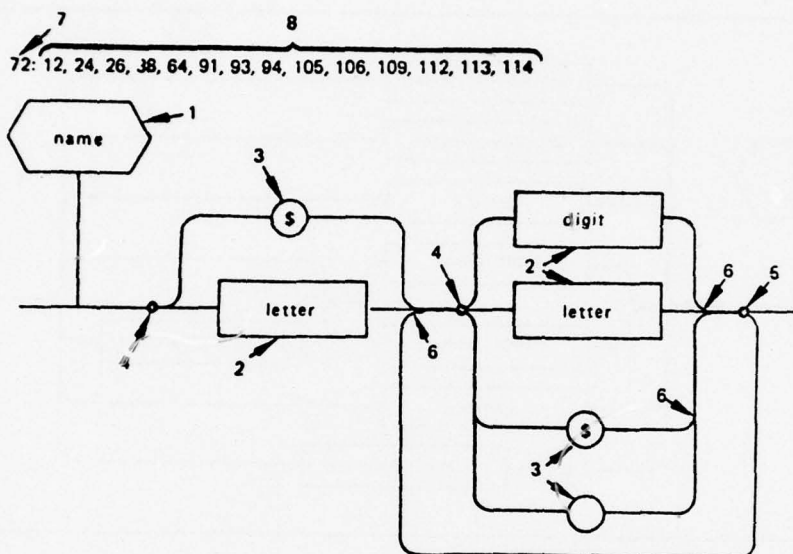
The assignment to LEFT and RIGHT will be traced for the odd iterations of LOOP while LOOP is less than 100.

APPENDIX A. SUMMARY SYNTAX

This appendix provides an alphabetized set of syntax statements which describe J73/I. A slightly different style of meta-language is used in this summary than in the remainder of the manual. It is hoped that this contrasting style when used in conjunction with the syntax of the main manual will ensure understanding of the language forms.

The meta-language used for this summary provides more of a flowchart description of the language elements which should be more usable as a reference section but would have been too cumbersome in the description.

The following example syntax statement will illustrate the meta-language:



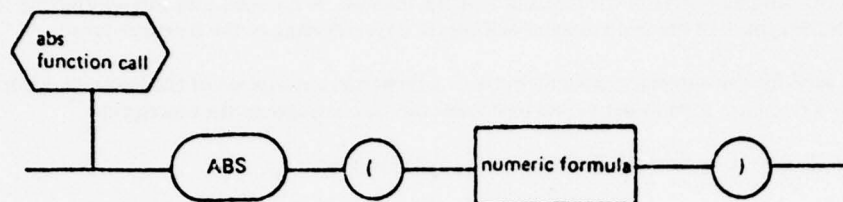
The rules for reading the meta-language are:

- Each meta-language statement defines some JOVIAL language element. The element being defined is enclosed in a hexagonal box (1).
- Elements used in the definition but defined by other meta-language statements are enclosed in rectangular boxes (2).
- Circular or oval boxes denote a JOVIAL source character or character sequence such as a key word (3).
- Flow begins left to right. A choice of paths is indicated by a dot (4, 5). Repetition is indicated by a line which doubles back (5). Backing up is not permitted where choices merge (6).
- The language elements are defined in alphabetical order. Each meta-language statement is given a sequential number (7). A cross-reference list is provided indicating the definitions which reference the element being defined (8).

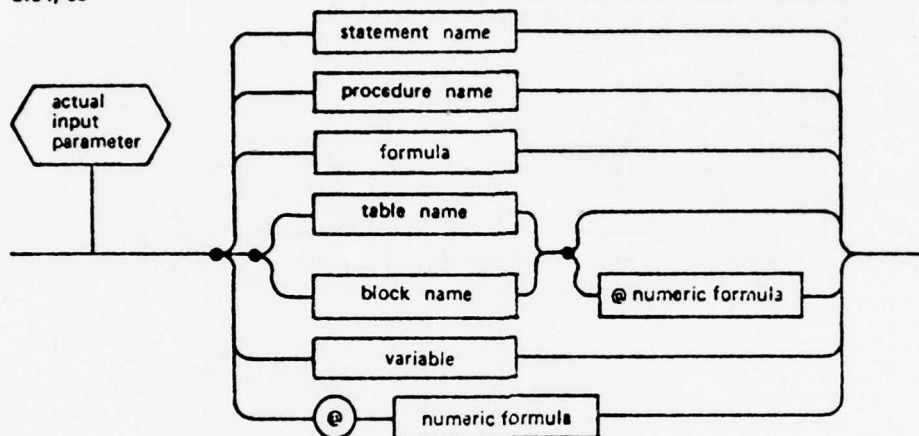
The above meta-language statement defines name to be a letter or dollar sign followed by any sequence of letters, digits, dollar signs, and primes.

The JOVIAL definitions which appear in this appendix are intended for use as a reference. To enhance its utility as a reference aid, certain JOVIAL elements which were defined independently in the text of this manual for descriptive purposes are combined into fewer, more meaningful definitions.

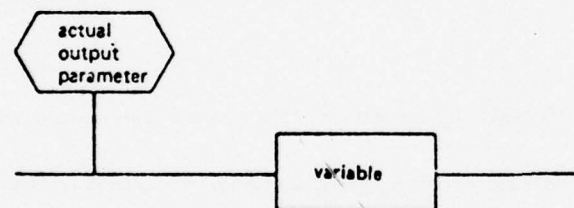
1:61



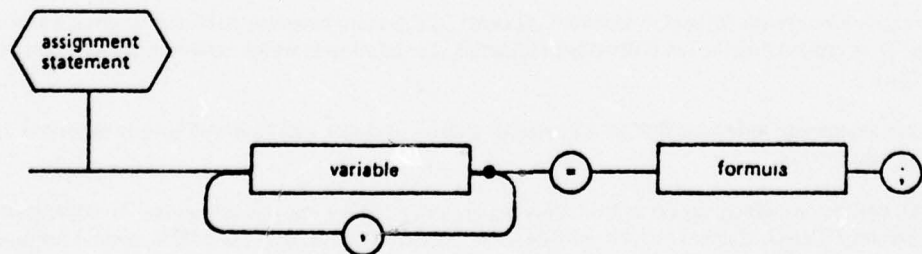
2:54, 89



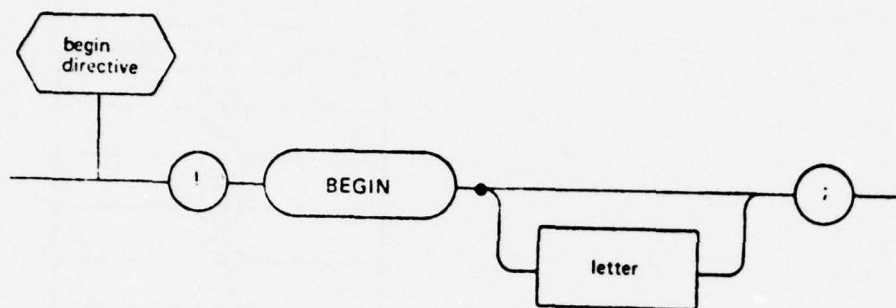
3:89



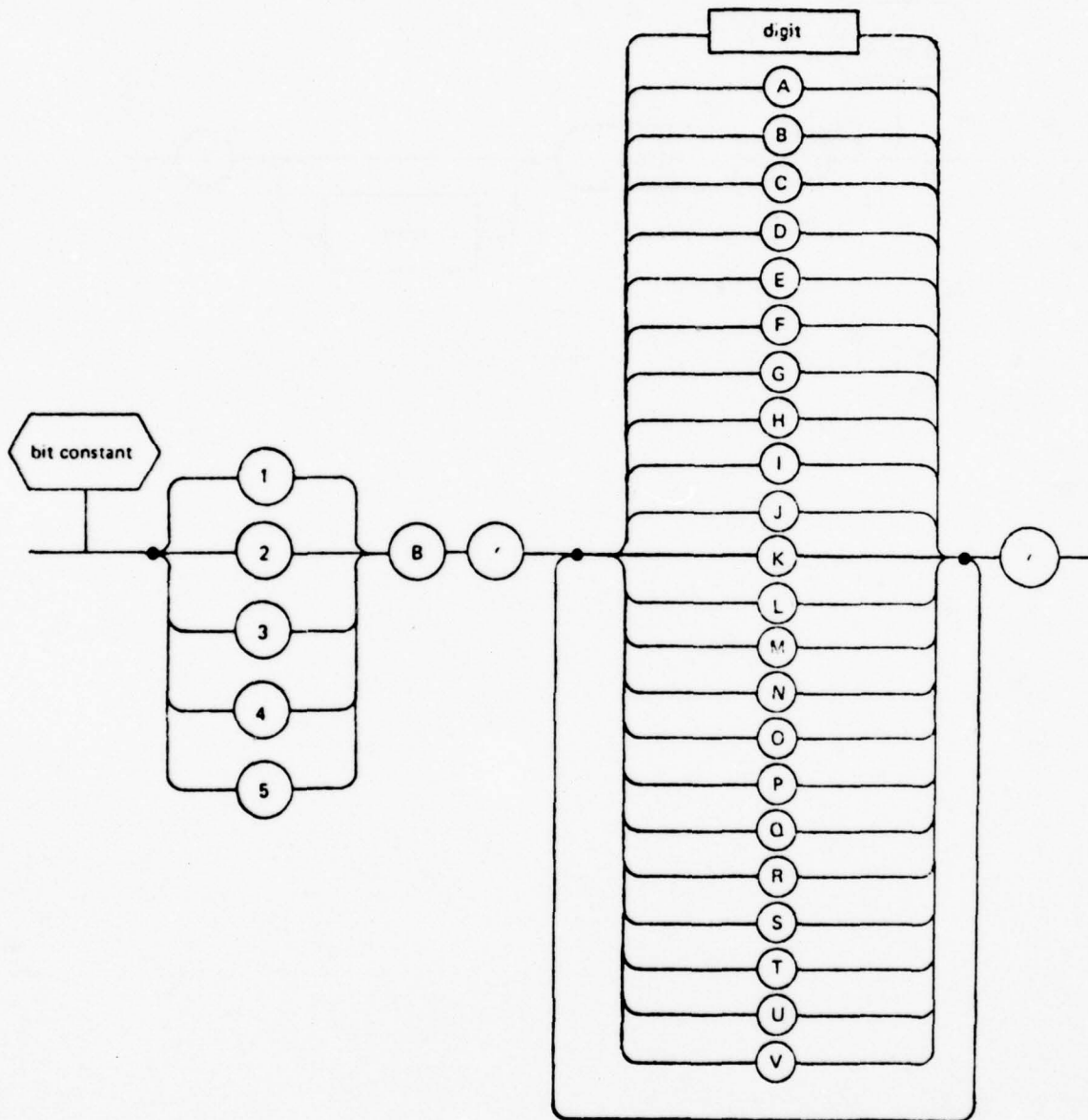
4:98



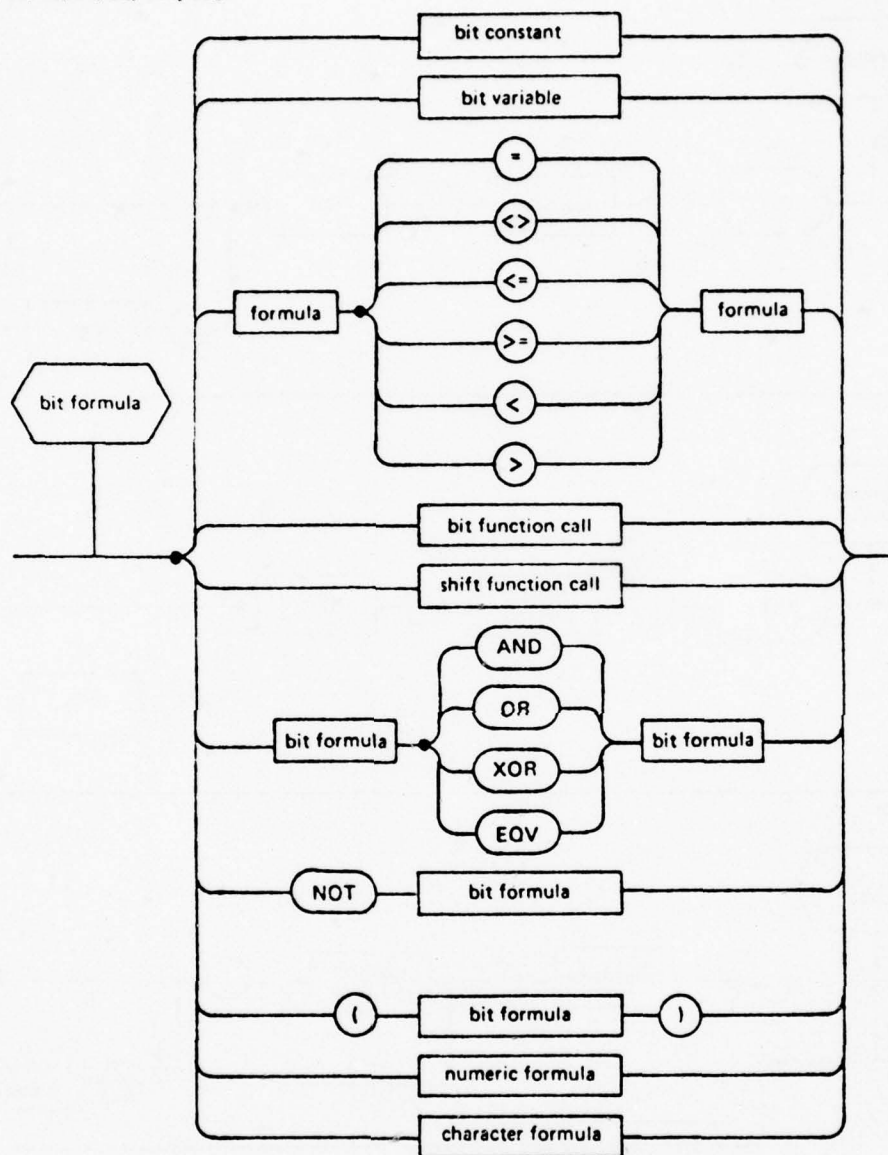
5:40



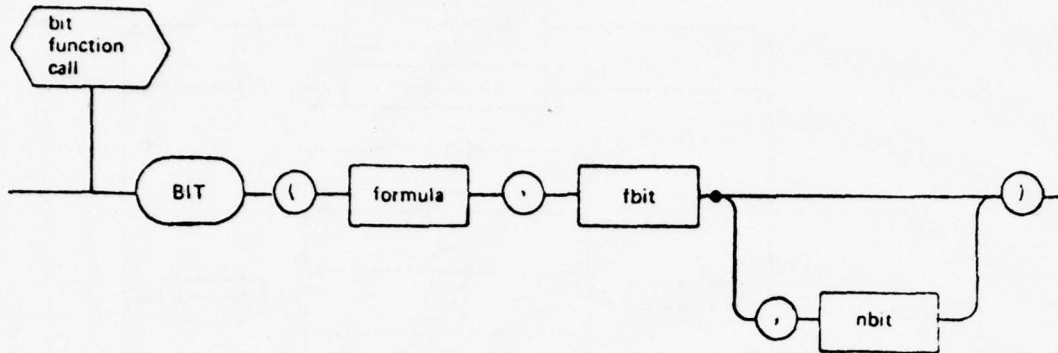
6: 7, 27, 57, 87



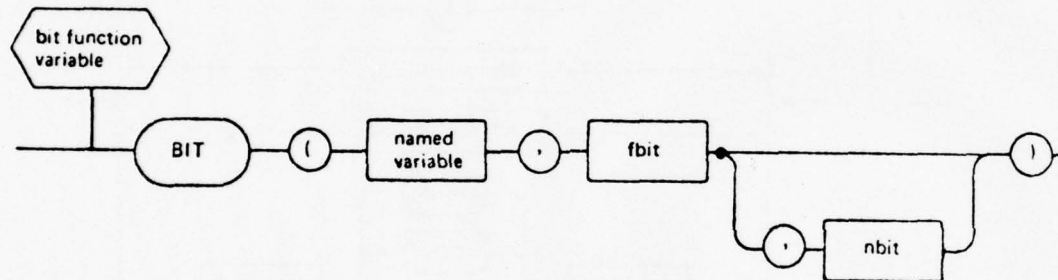
7: 7, 49, 53, 56, 81, 96, 114, 116



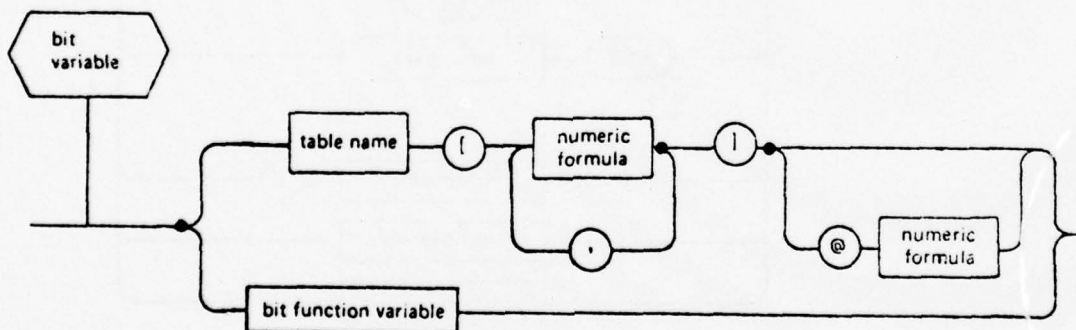
8: 7, 61



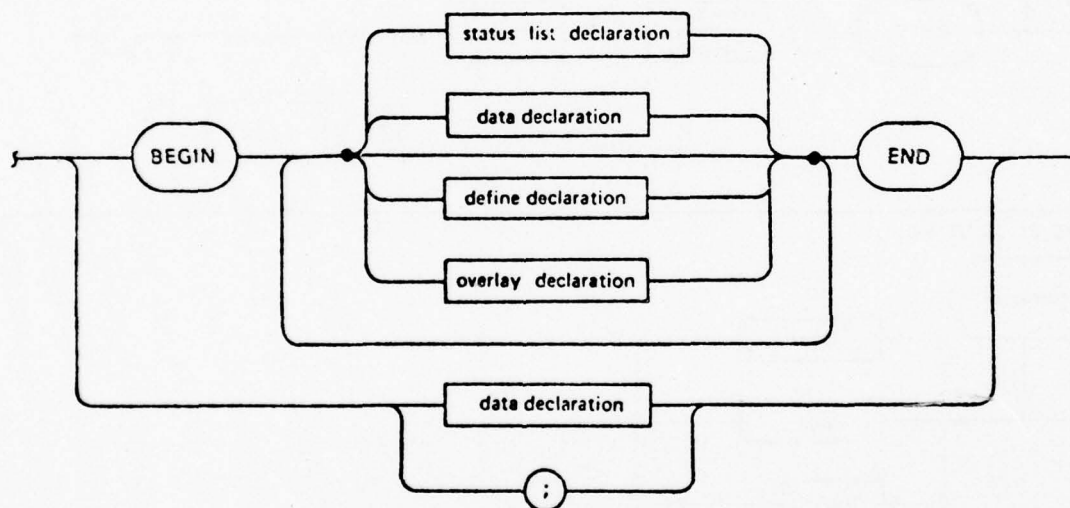
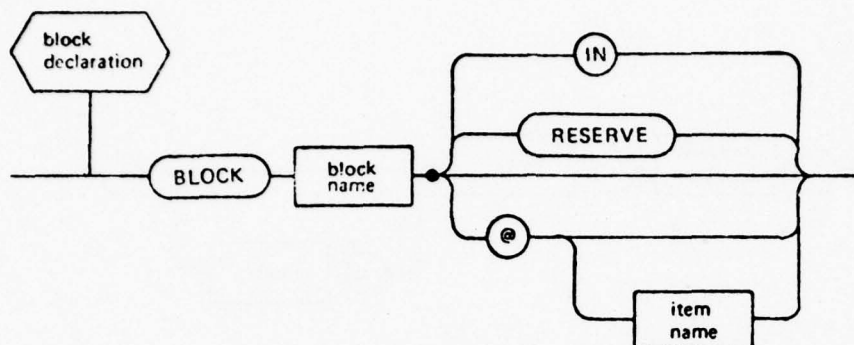
9: 10



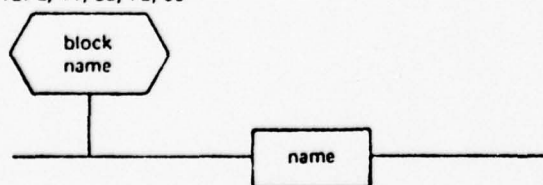
10: 7, 115



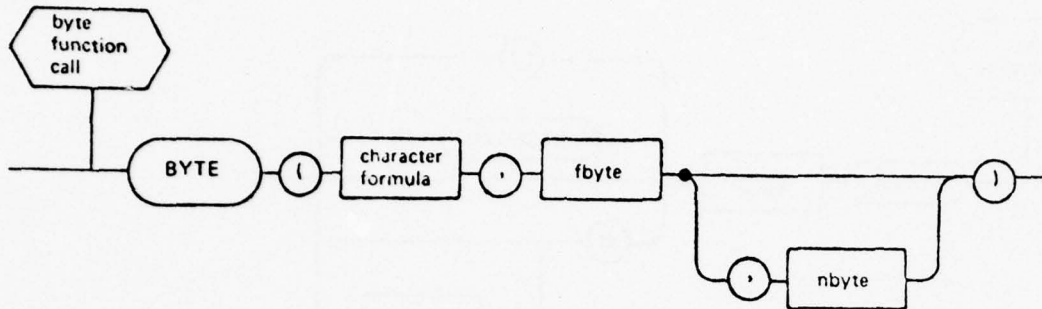
11: 31



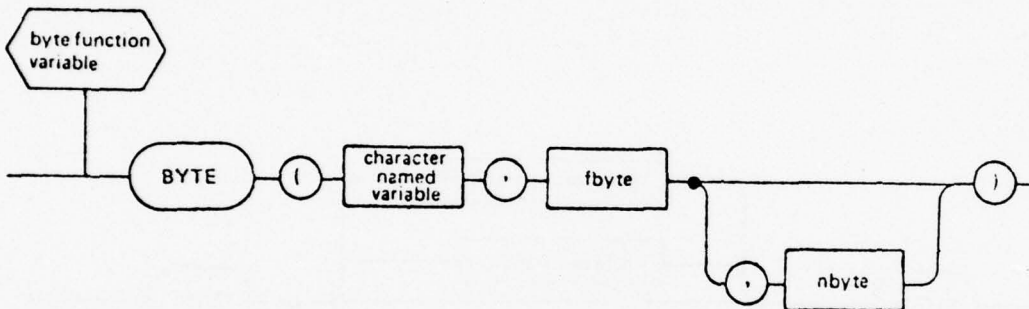
12: 2, 11, 32, 72, 99



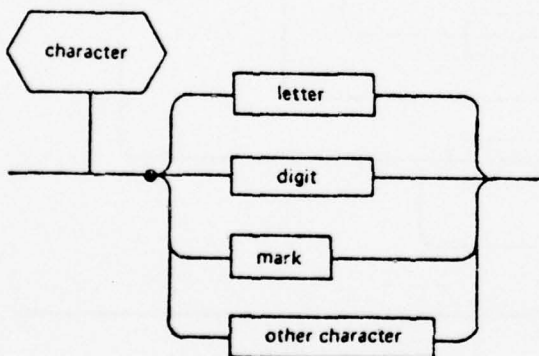
13: 18, 61, 115



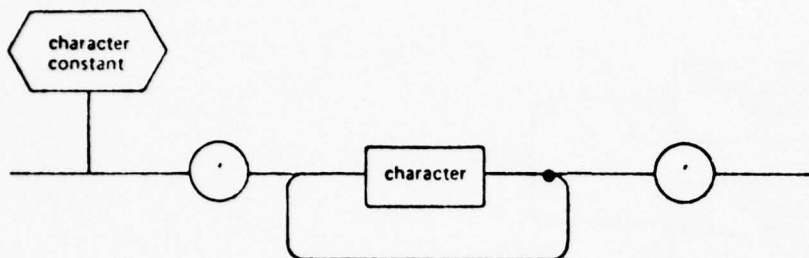
14: 20



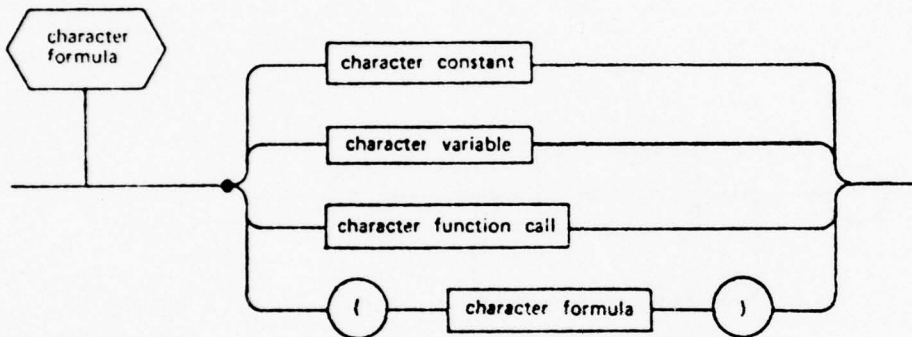
15: 16, 21, 36, 37, 57



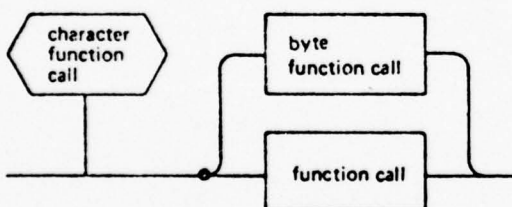
16: 17, 25, 27, 30



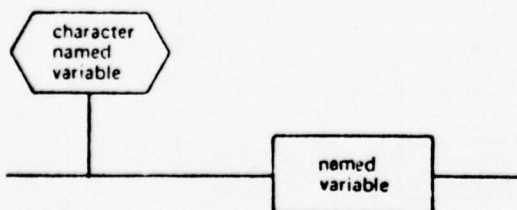
17: 7, 13, 17, 53



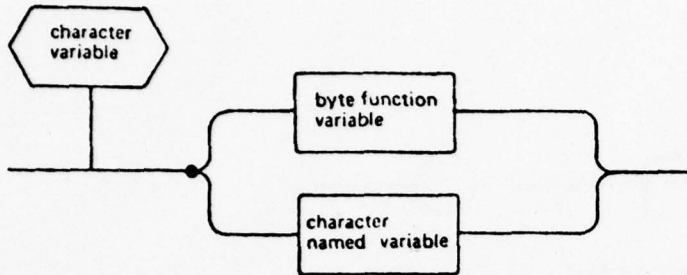
18: 17



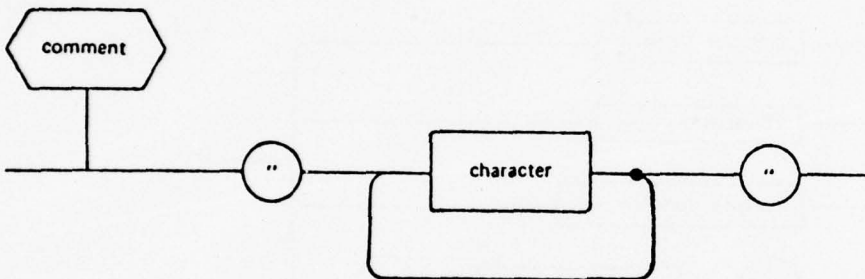
19: 14, 20



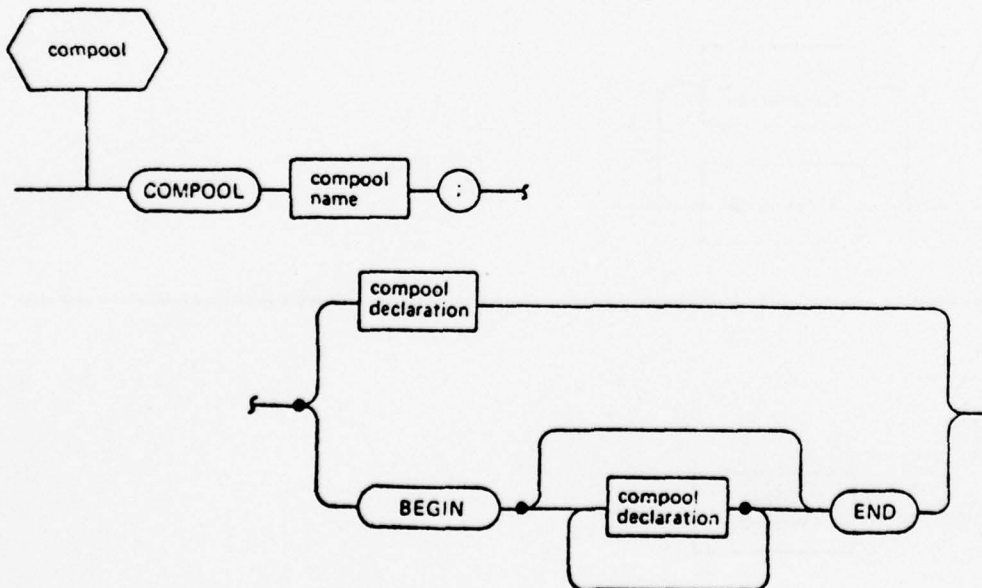
20: 17



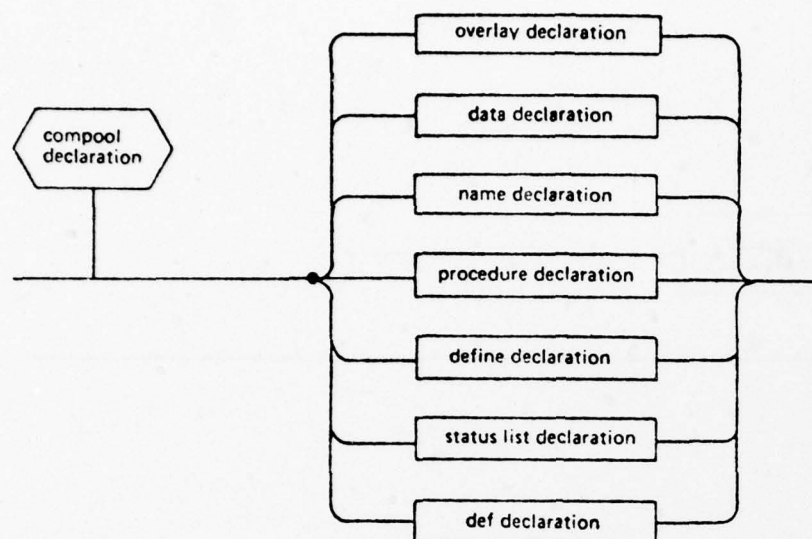
21:



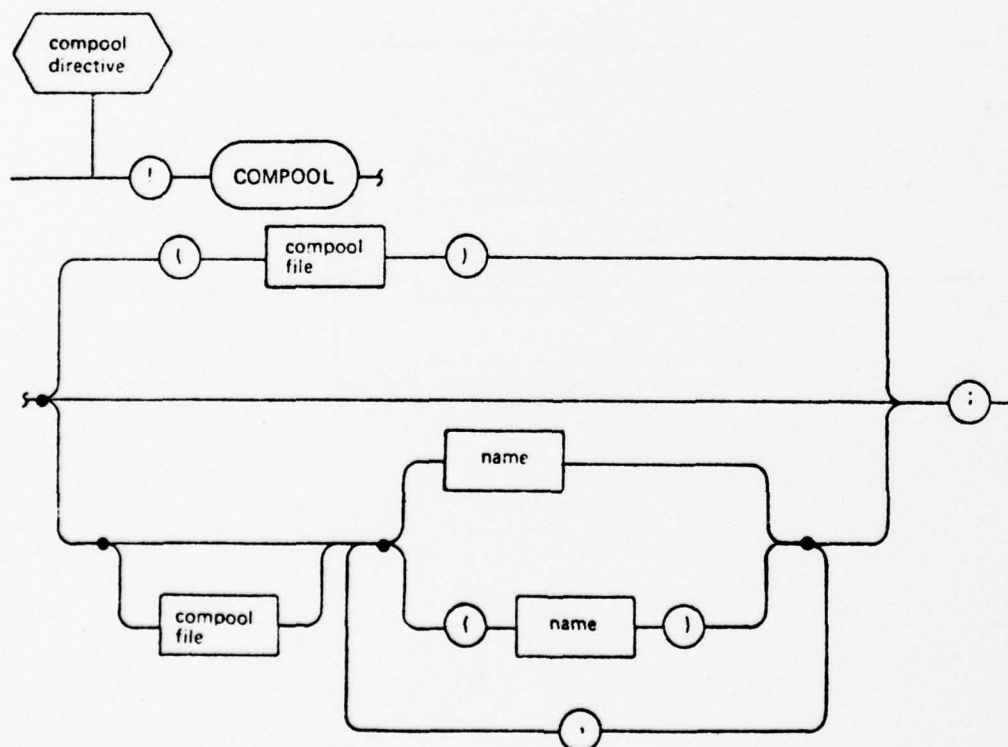
22: 65



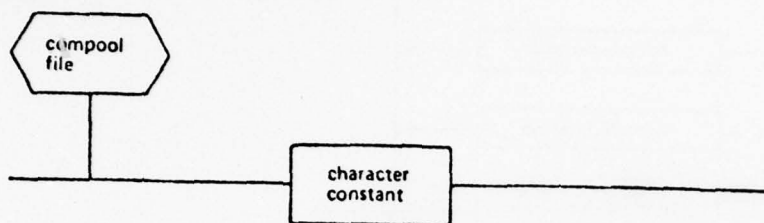
23: 22



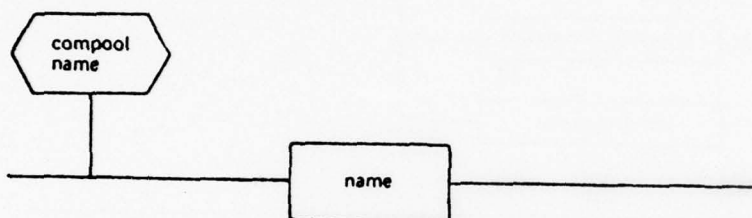
24: 40



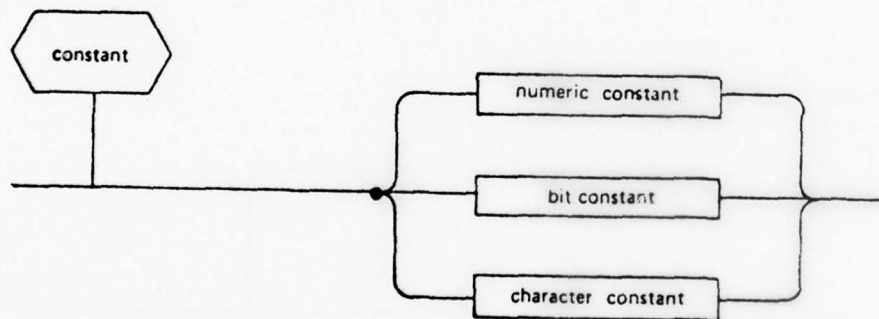
25: 24



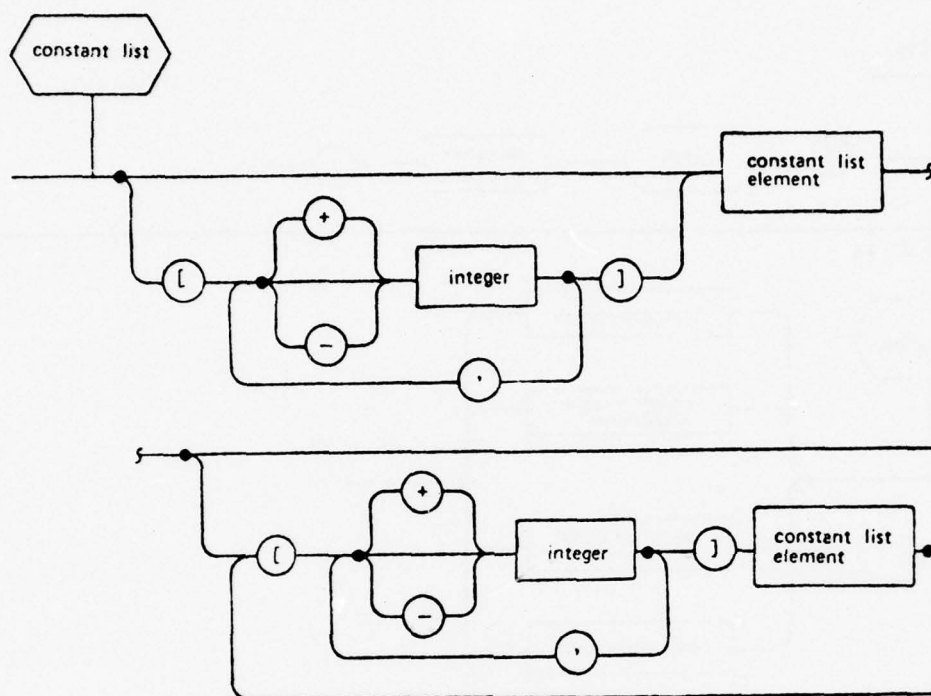
26: 22



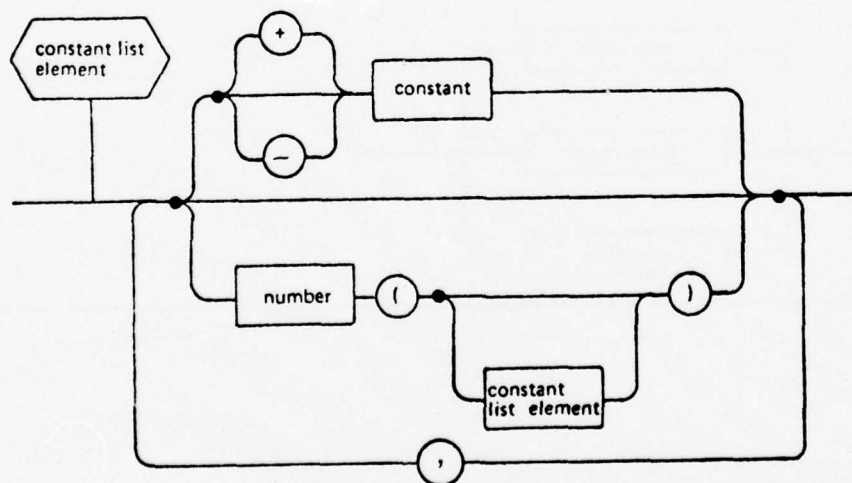
27: 29, 62, 90



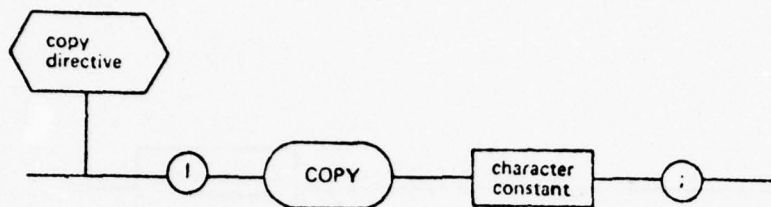
28: 85, 86, 102, 103



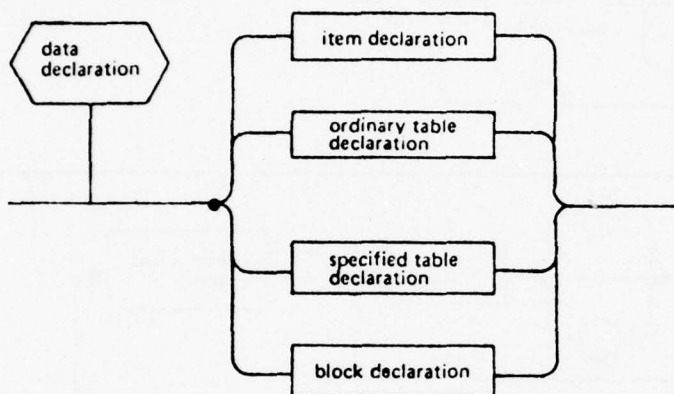
29: 28, 29



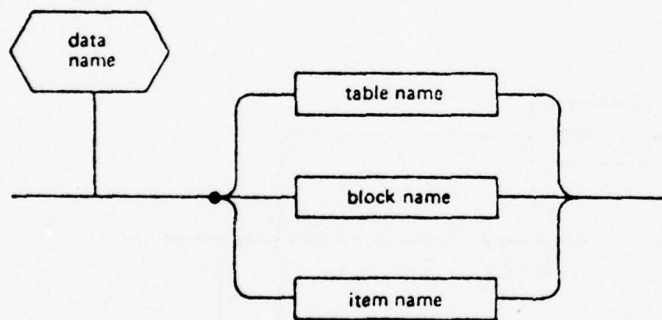
30: 40



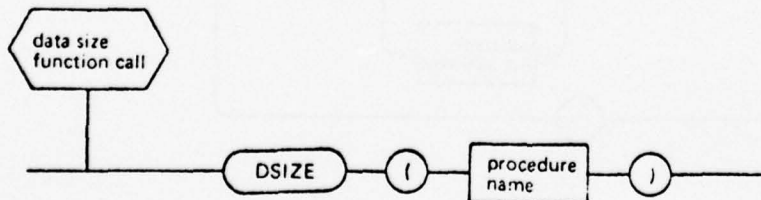
31: 11, 23, 34, 35, 43



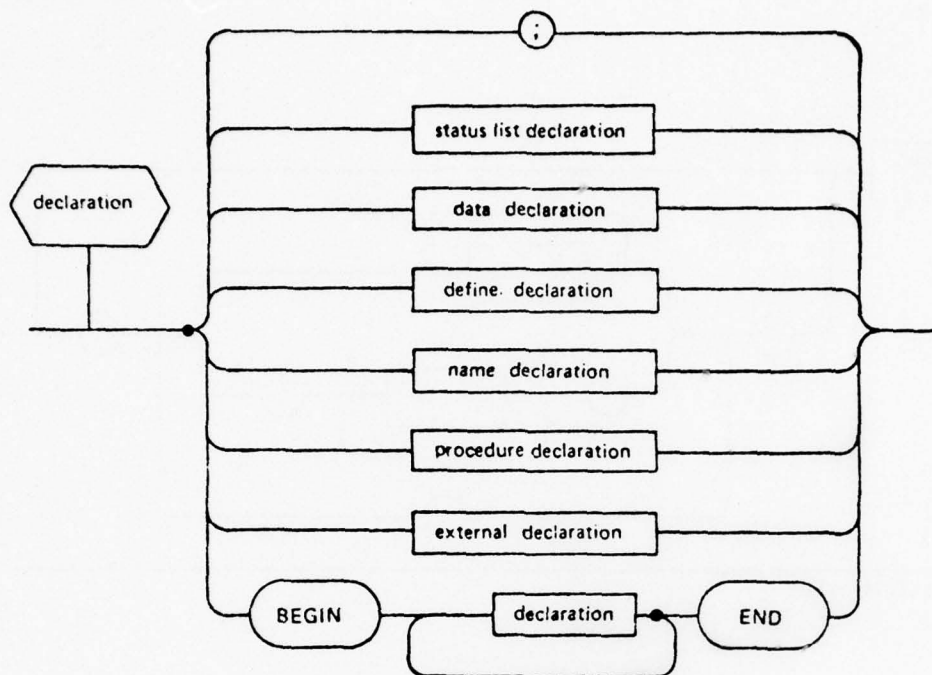
32: 51, 88



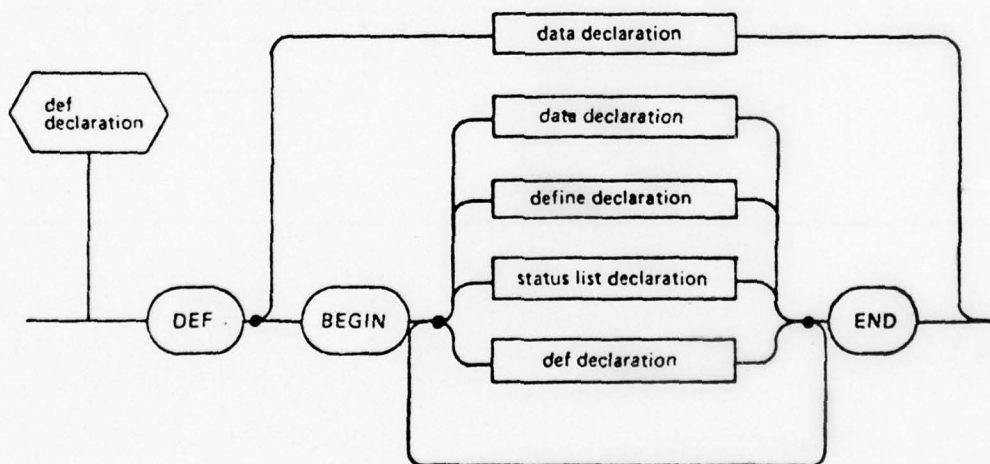
33: 61



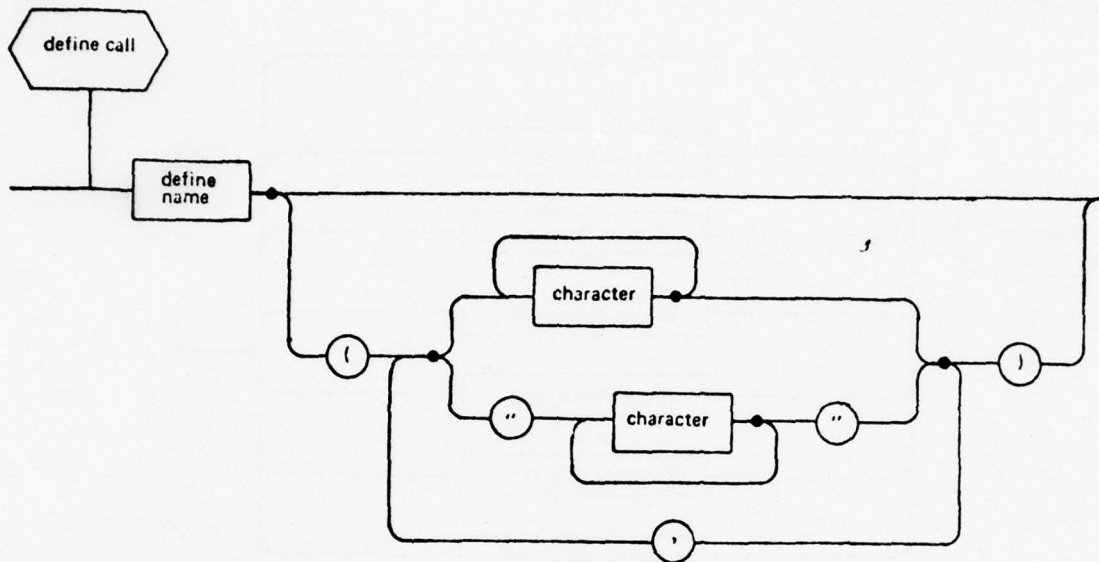
34: 34, 90, 92, 104



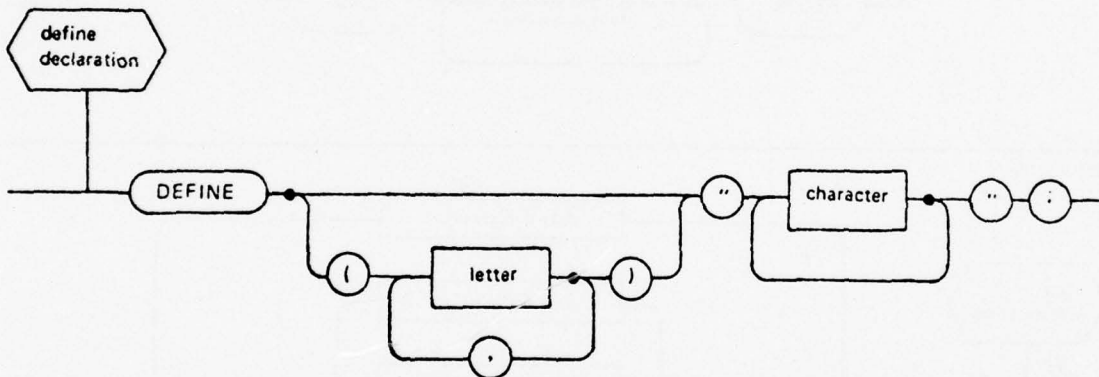
35: 23, 35



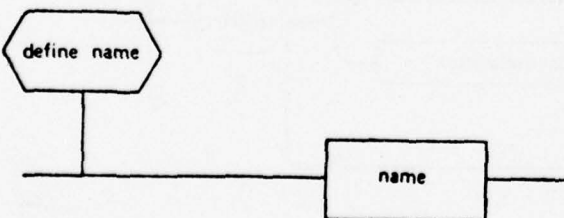
36:



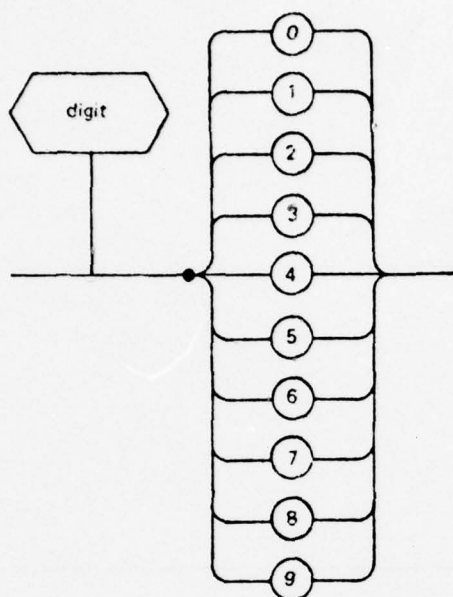
37: 11, 23, 34, 35, 43, 84, 101



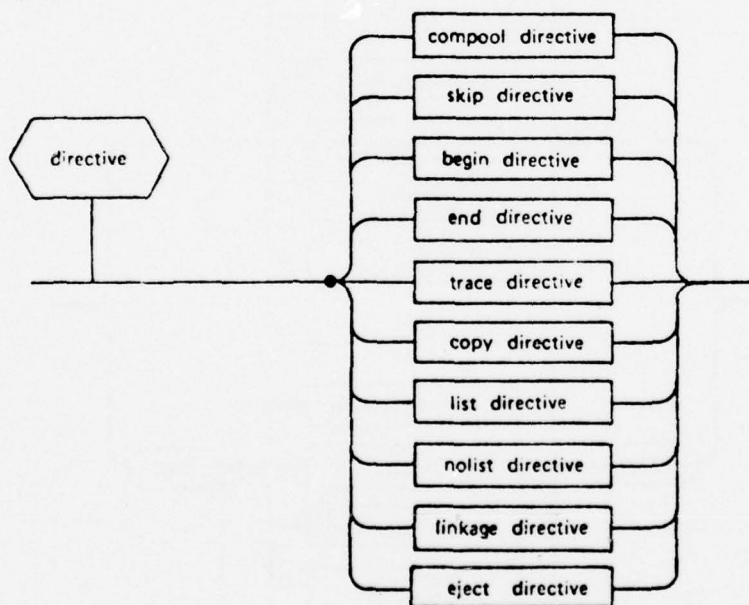
38: 36



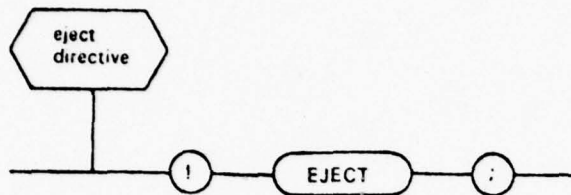
39: 6, 15, 47, 72, 78



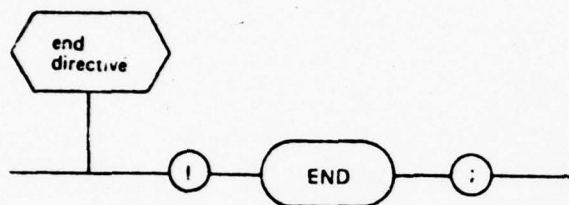
40: 65



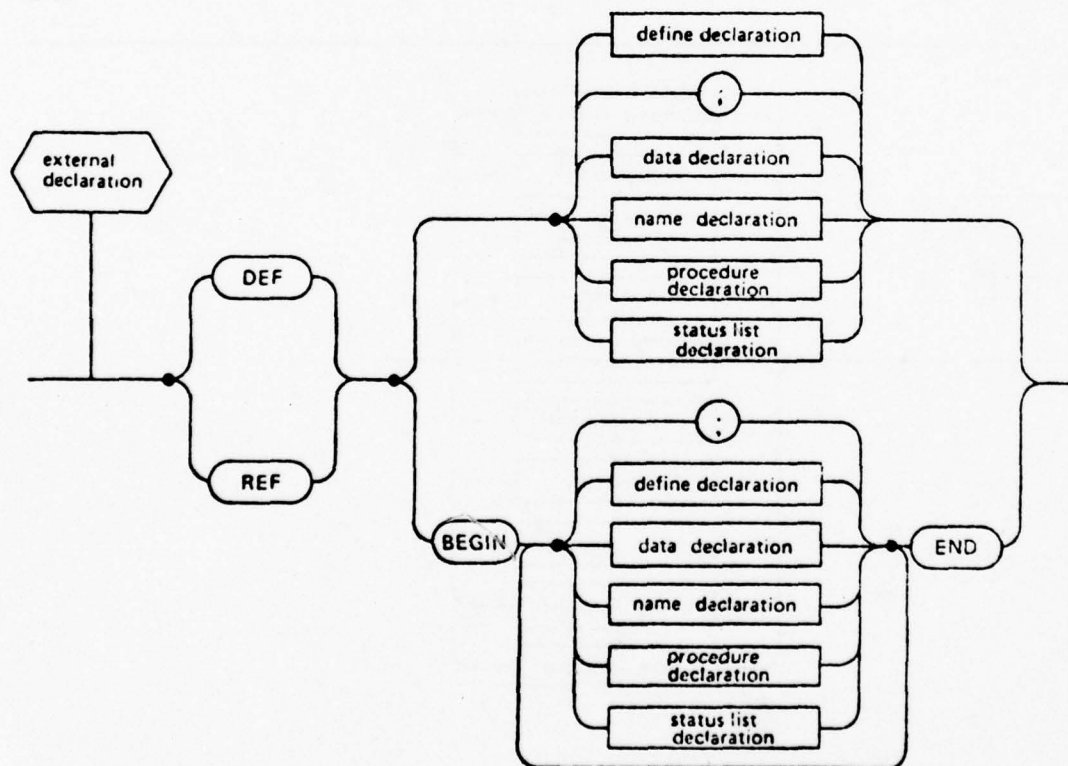
11: 40



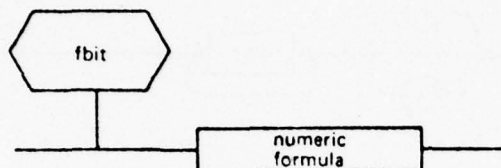
42: 40



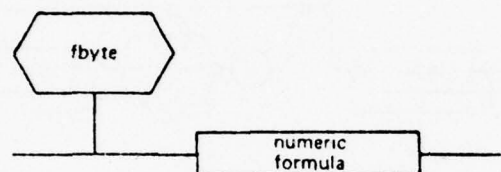
43: 34



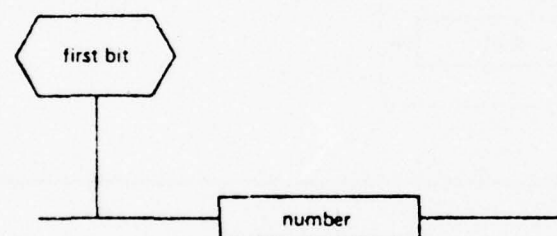
44: 8, 9



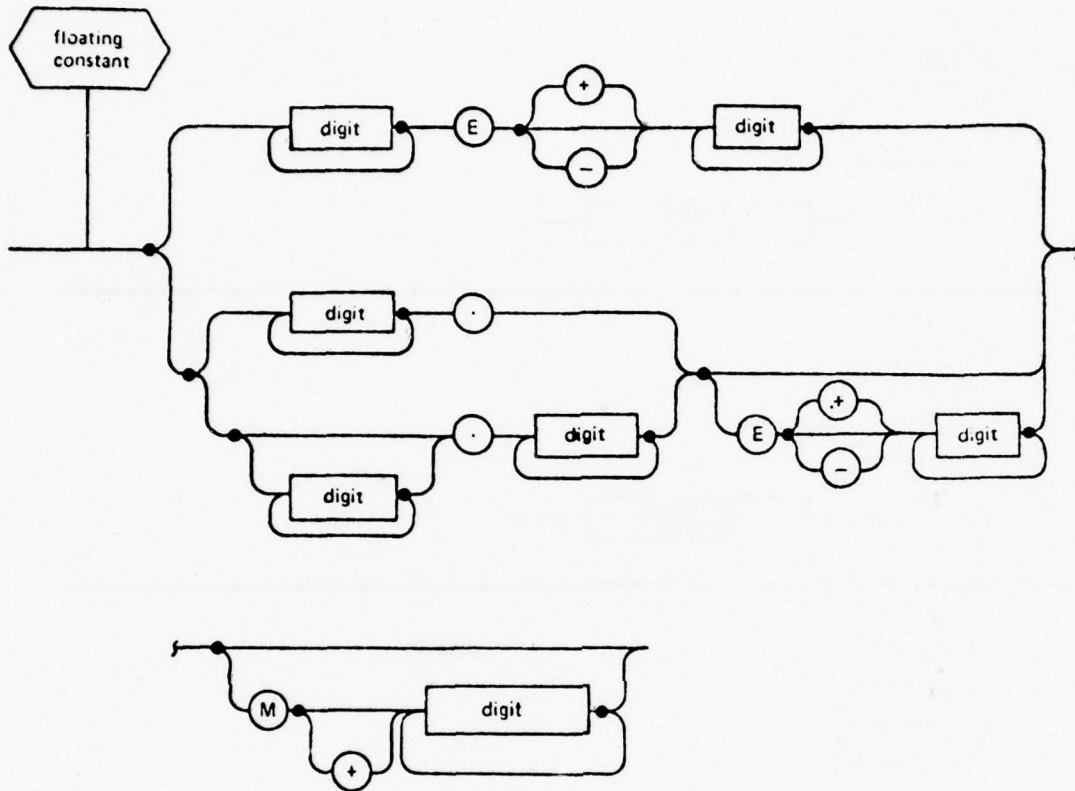
45: 13, 14



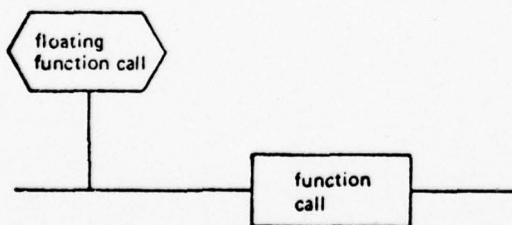
46: 102, 103



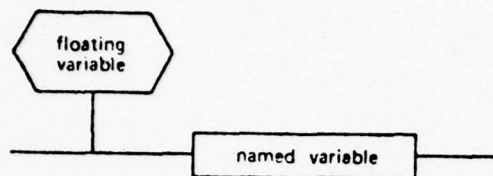
47: 80



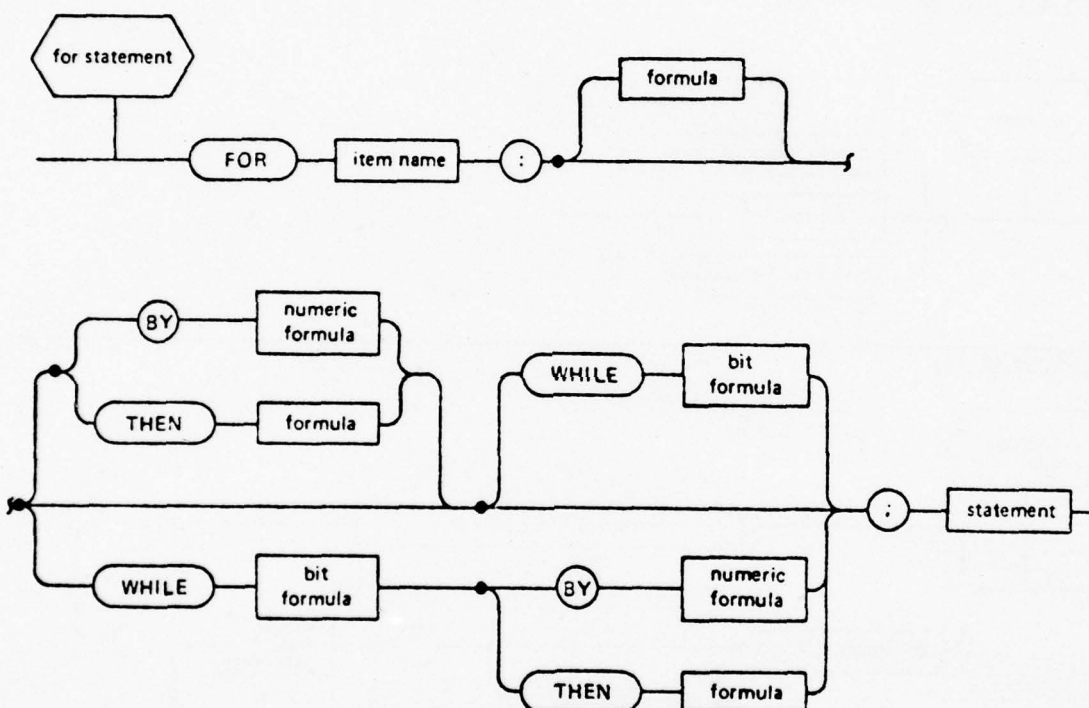
48: 82



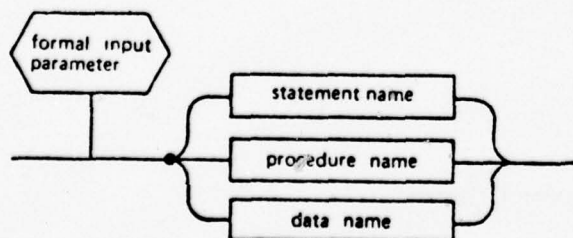
49: 83



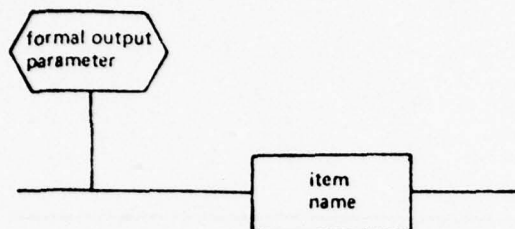
50: 98



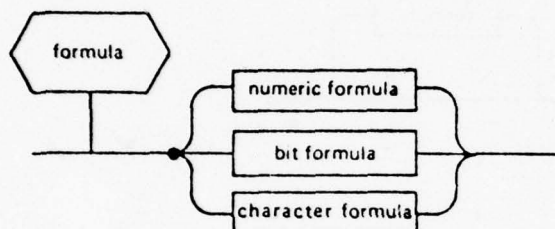
51: 90



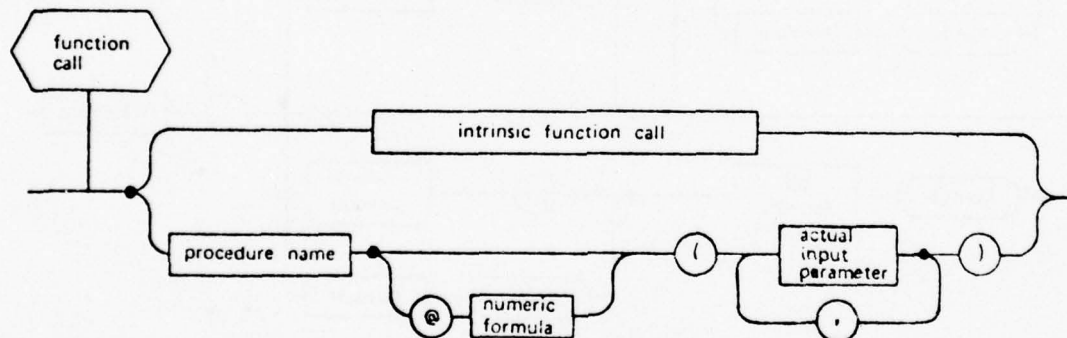
52: 91



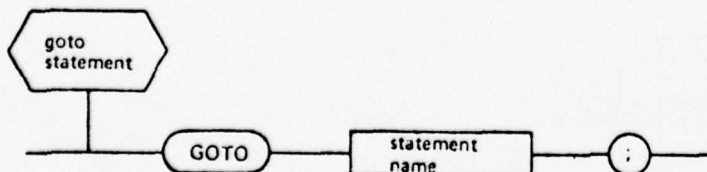
53: 2, 4, 7, 8, 50, 99



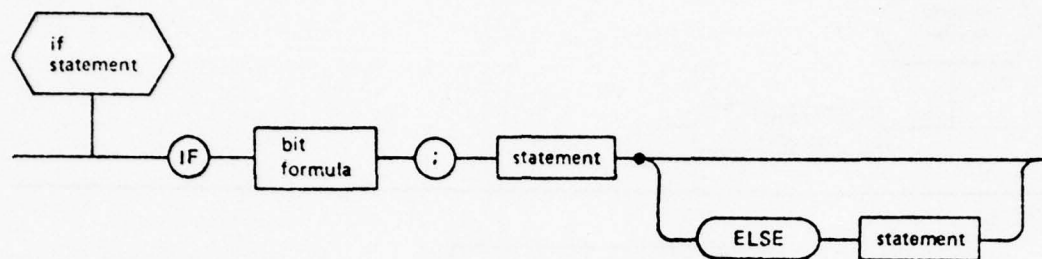
54: 18, 48, 59, 107



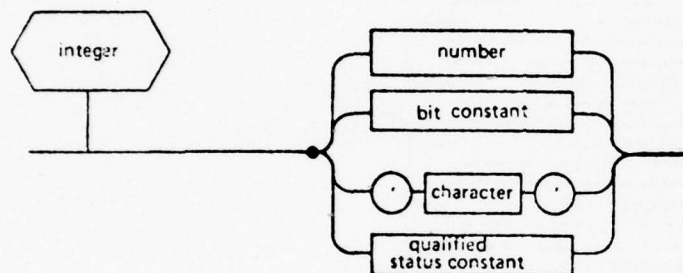
55: 98



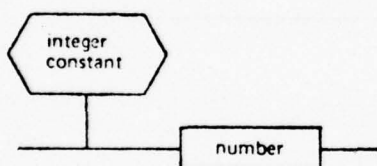
56: 98



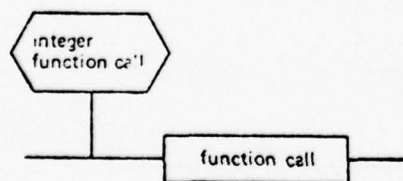
57: 28, 79, 85, 102, 108, 111



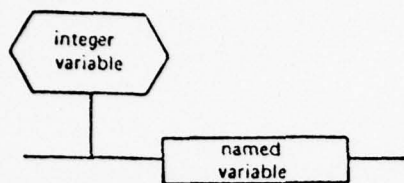
58: 80



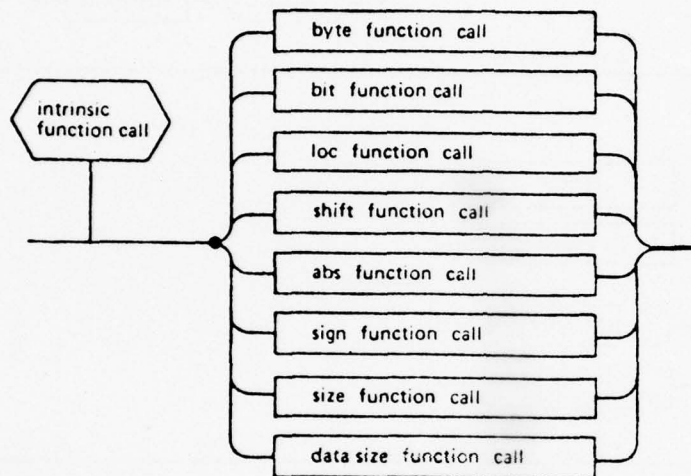
59: 82



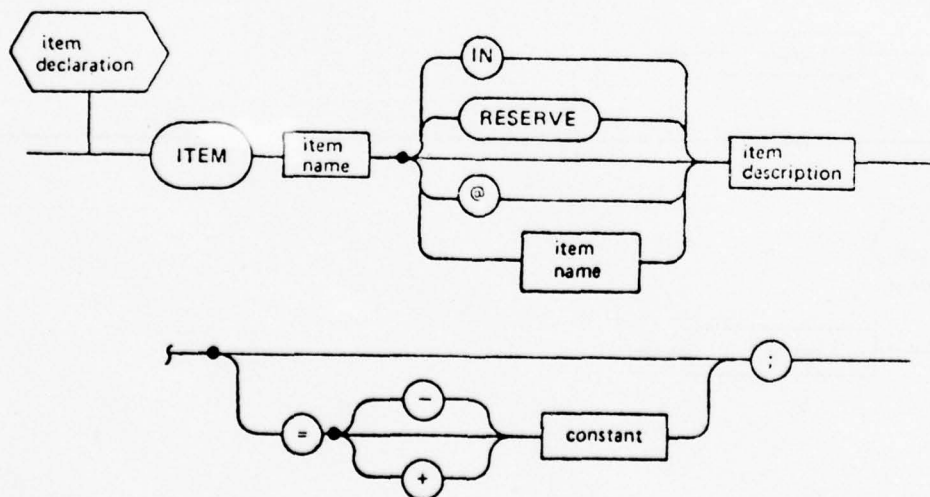
60: 83



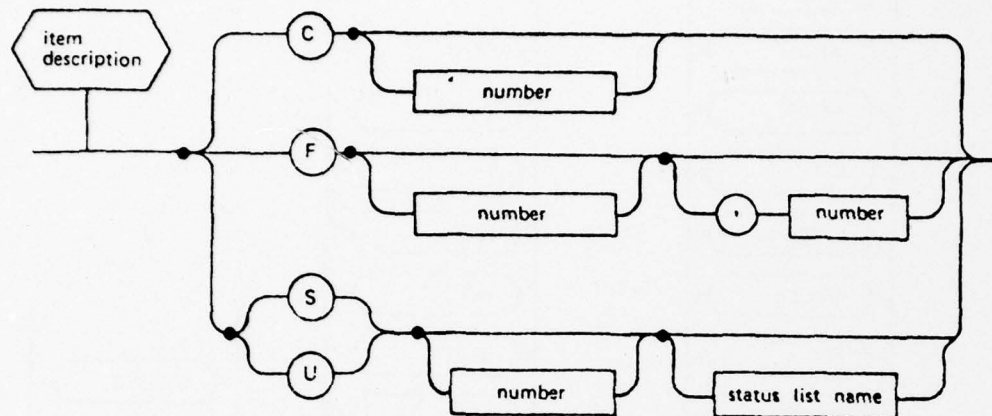
61: 54



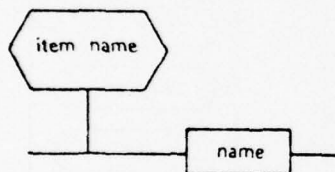
62: 31



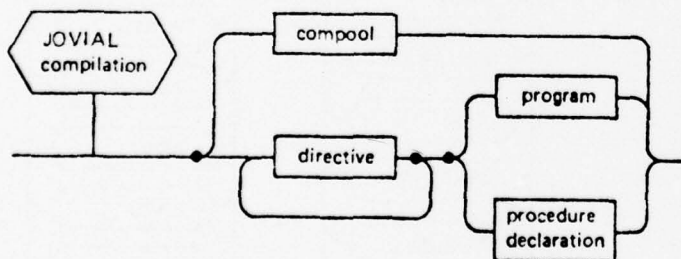
63: 62, 85, 86, 90, 102, 103



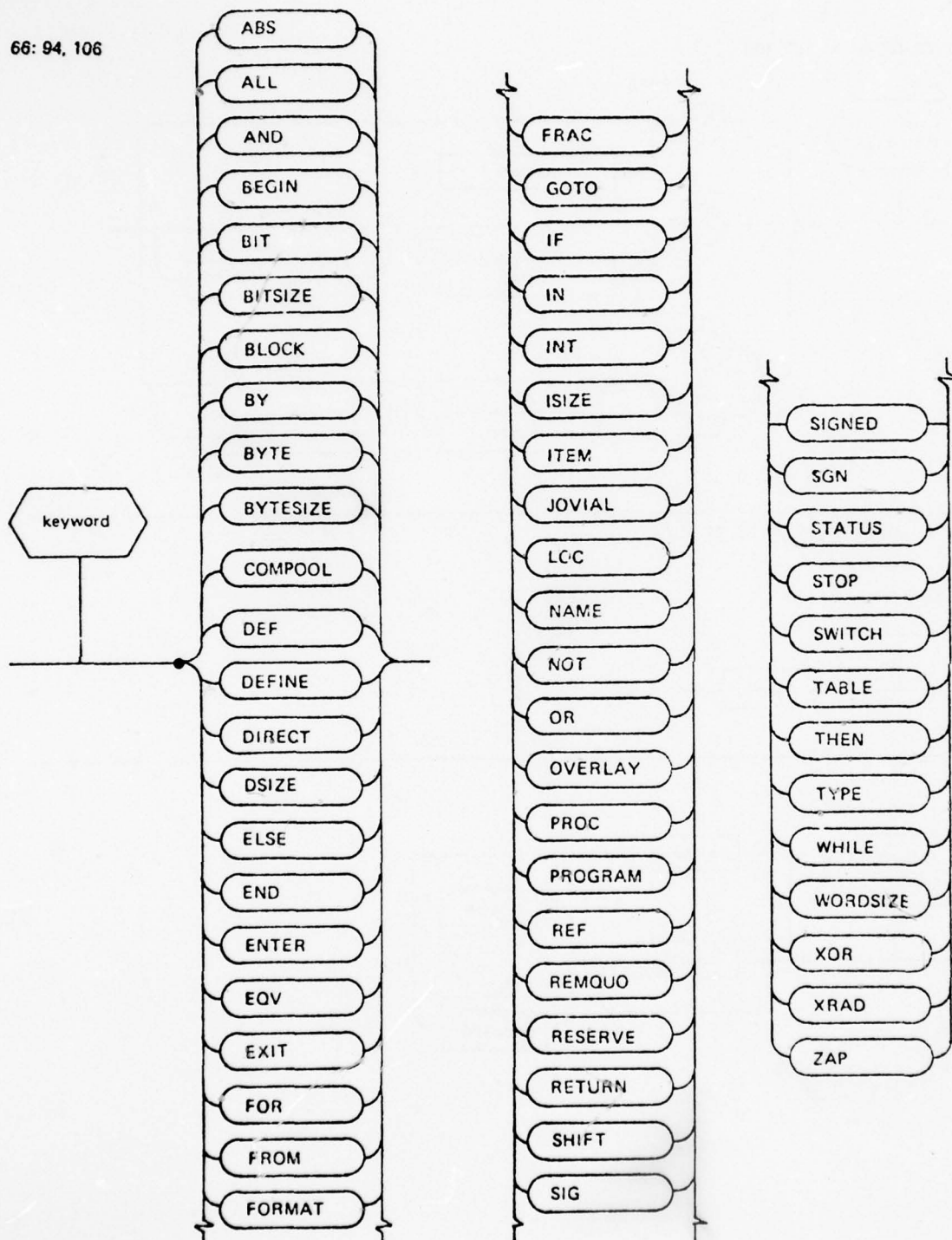
64: 11, 32, 50, 52, 62, 74, 85, 94, 102



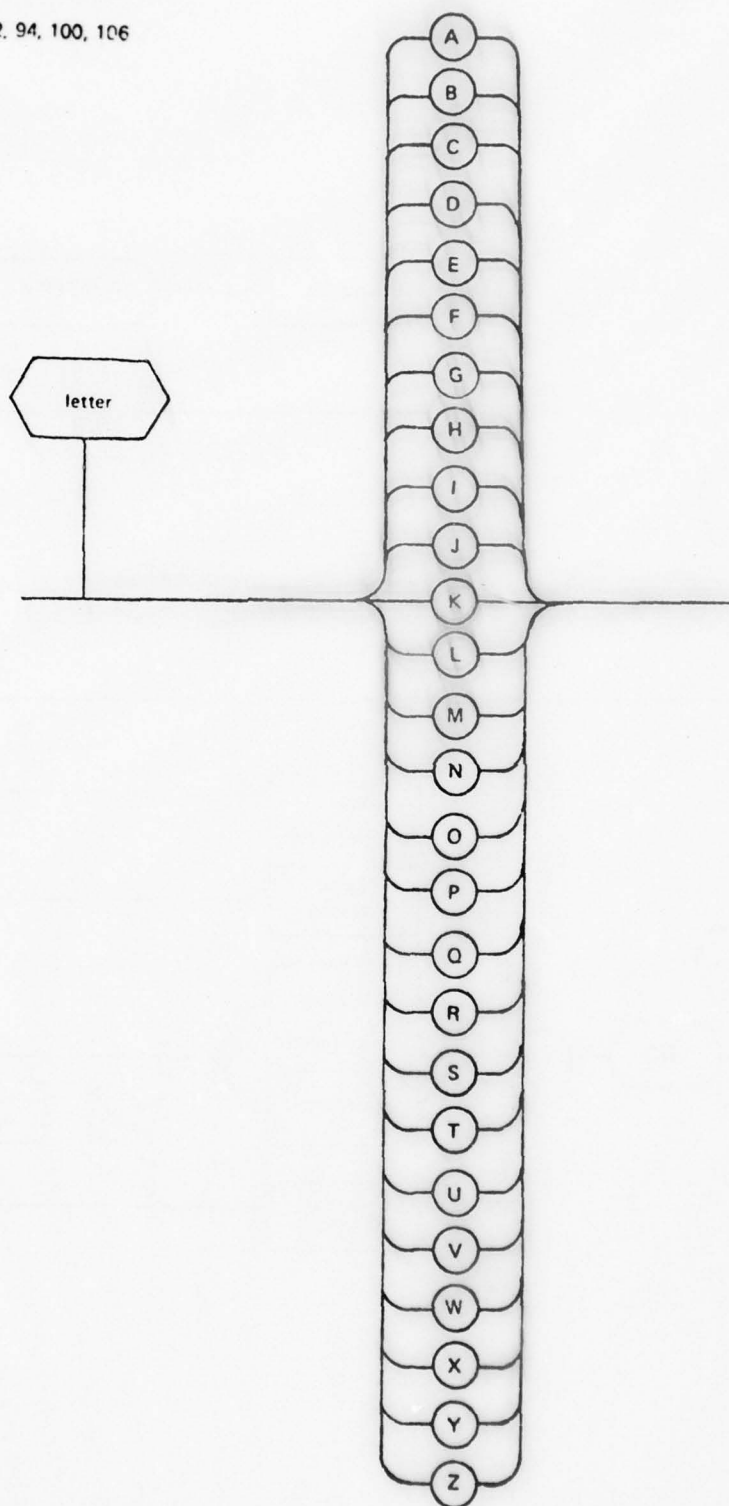
65:



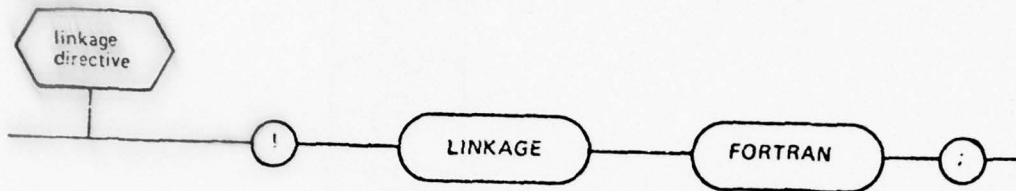
66: 94, 106



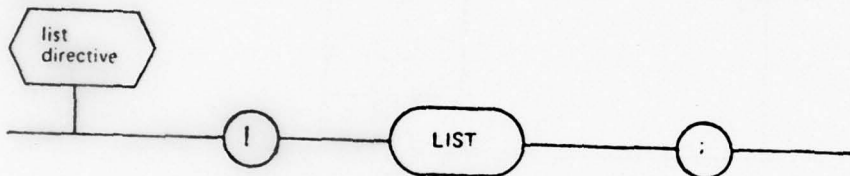
67: 5, 15, 37, 72, 94, 100, 106



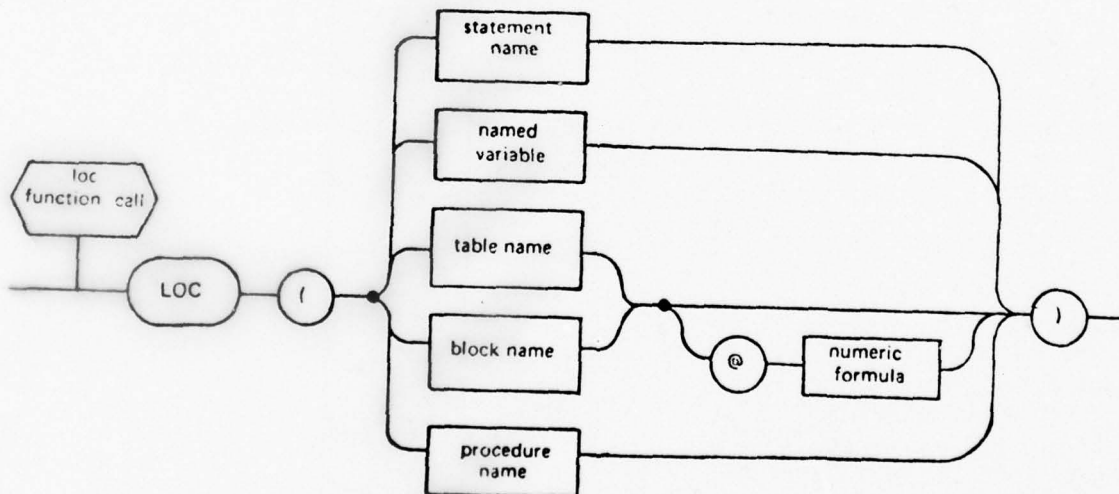
68: 40, 90



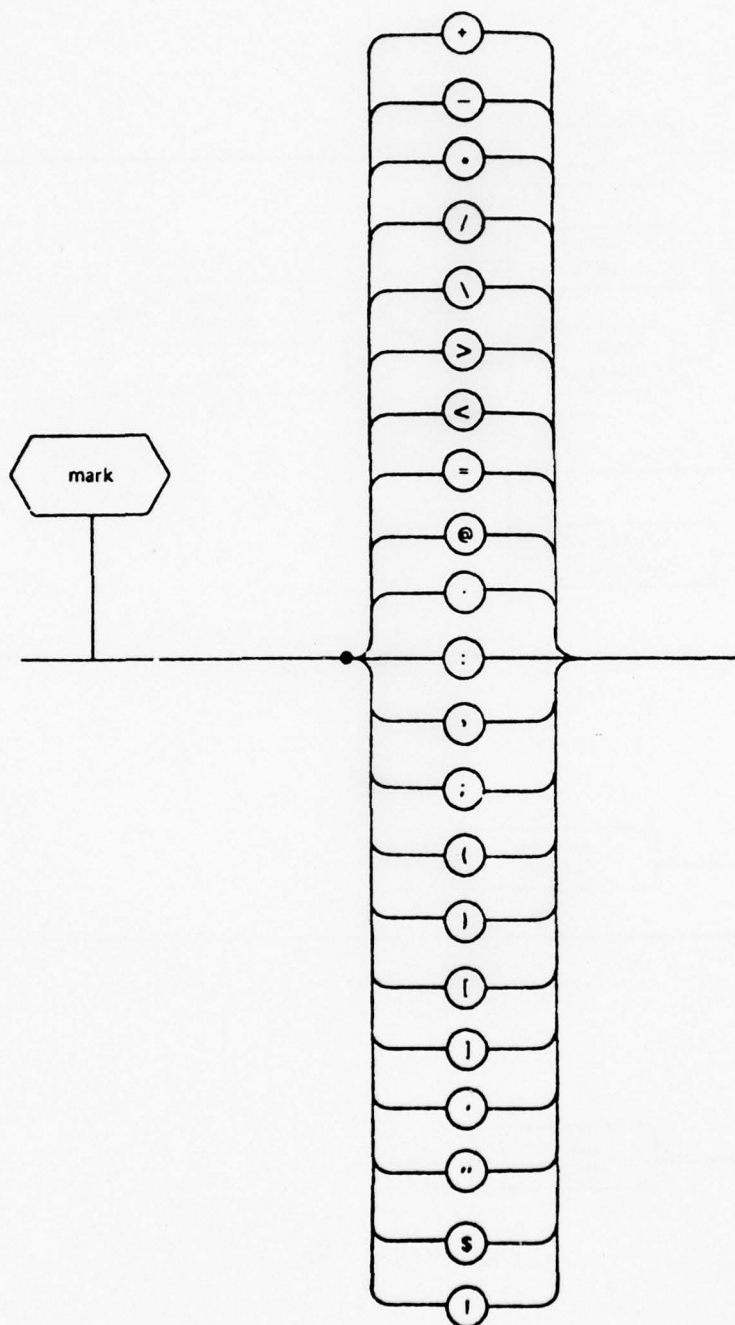
69: 40



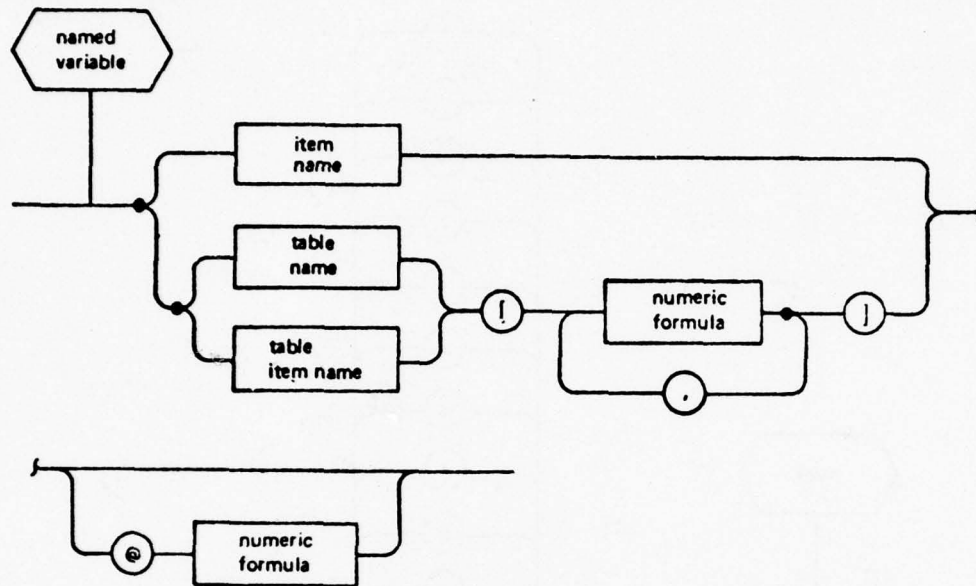
70: 61



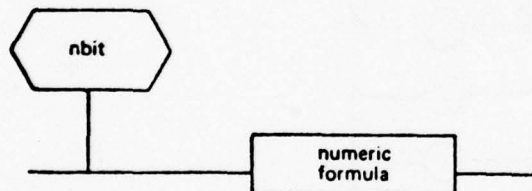
71: 15



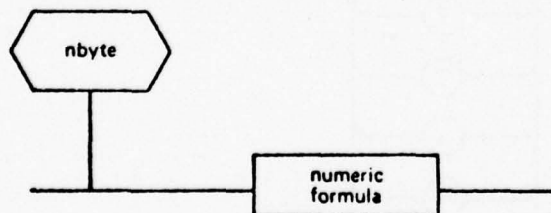
74: 9, 19, 49, 60, 70, 115



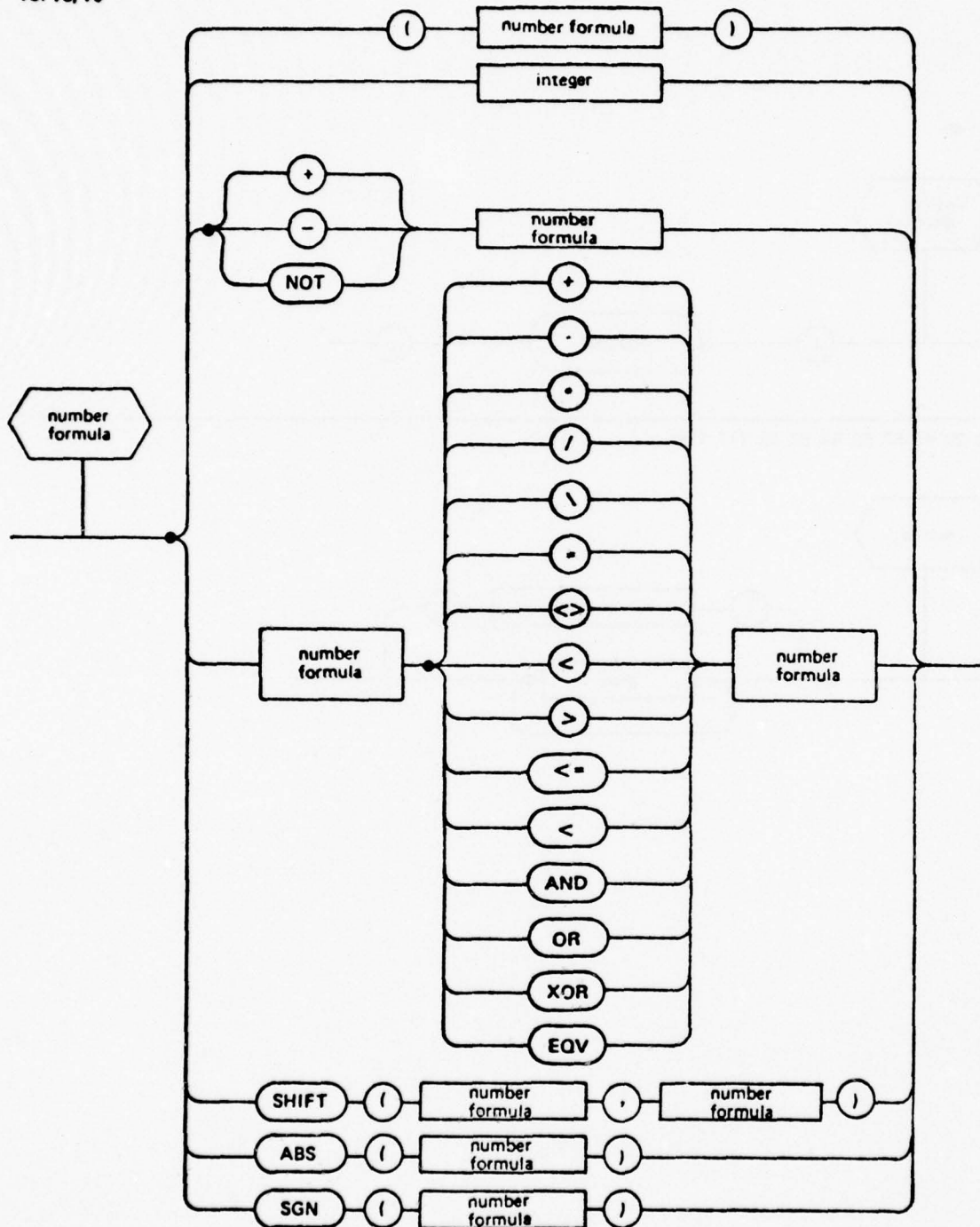
75: 8, 9



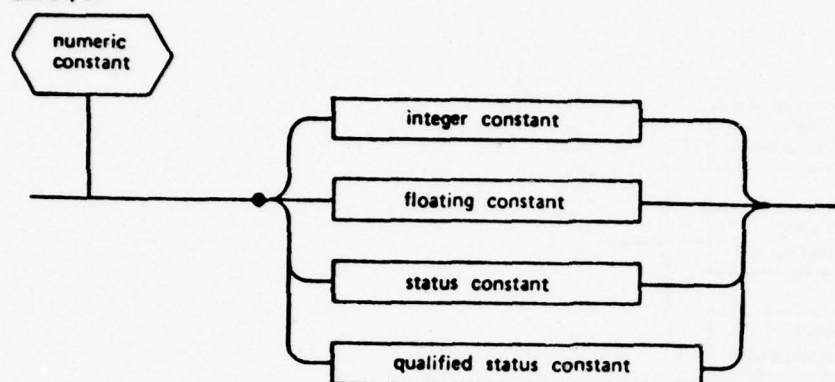
76: 13, 14



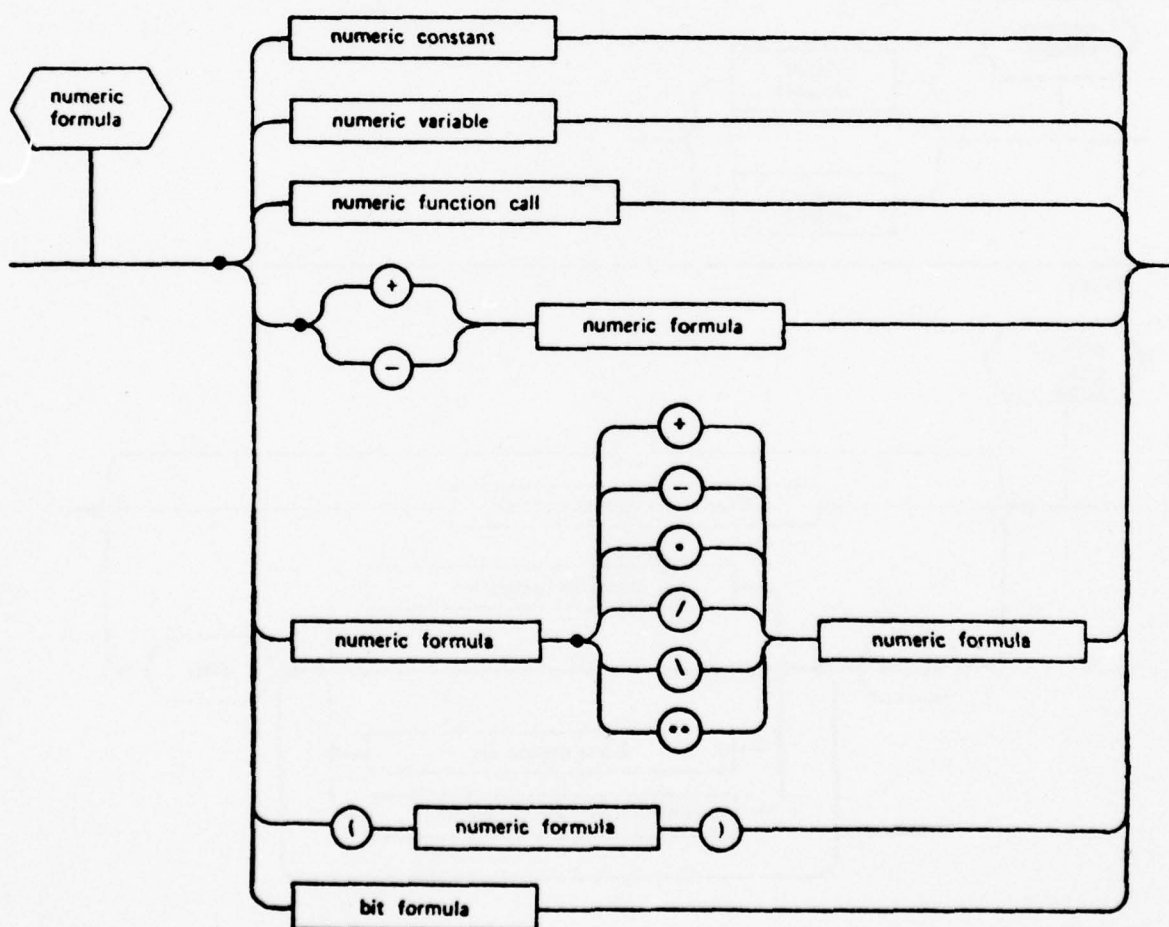
79: 78, 79



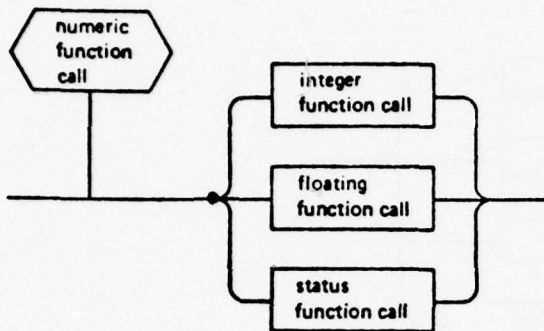
80: 27, 81



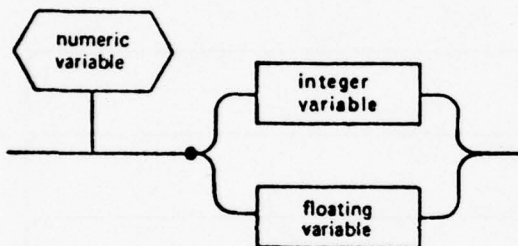
81: 1, 2, 7, 10, 44, 45, 50, 53, 54, 70, 74, 75, 76, 81, 89, 96, 97, 111



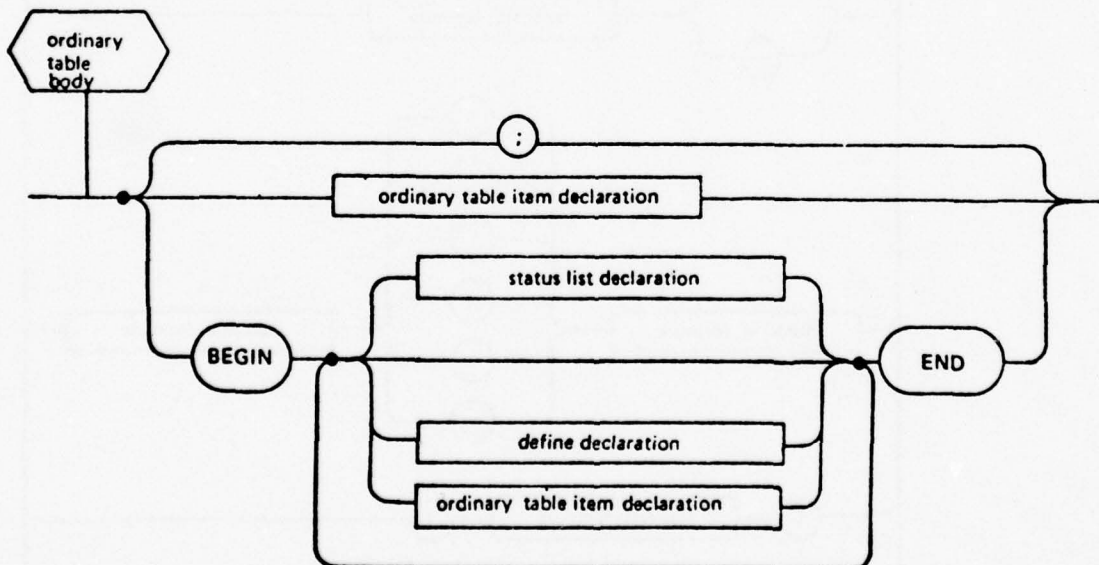
82: 81



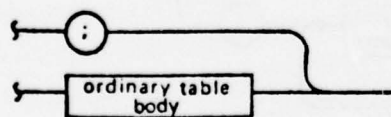
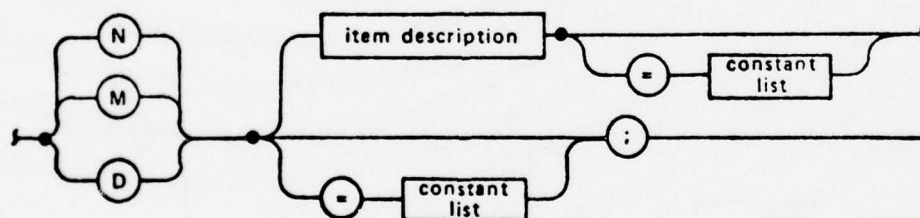
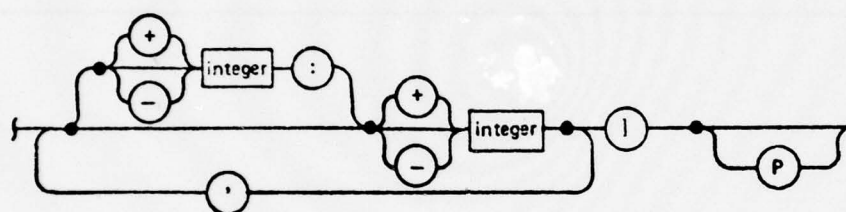
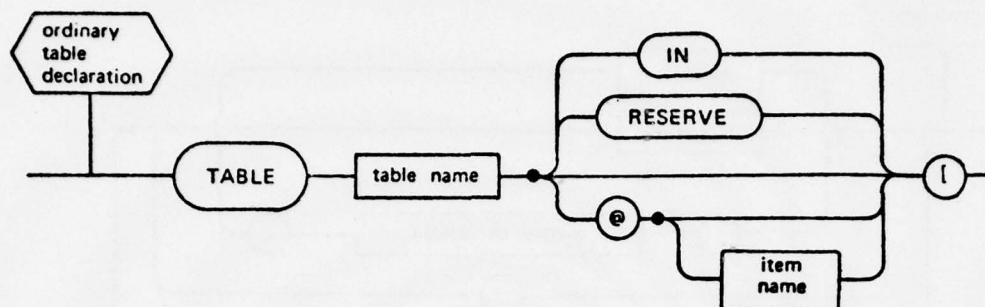
83: 81



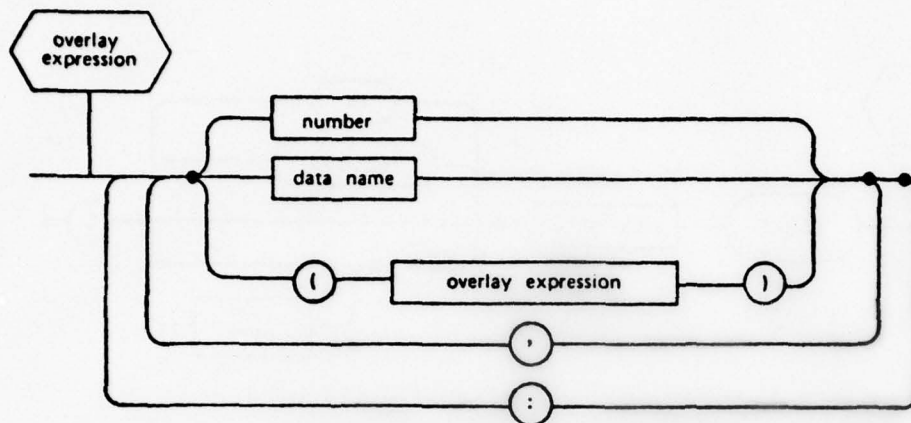
84: 85



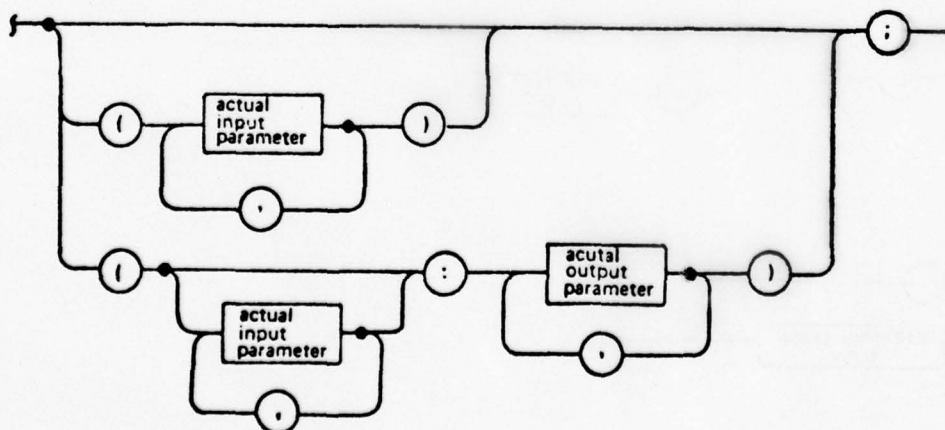
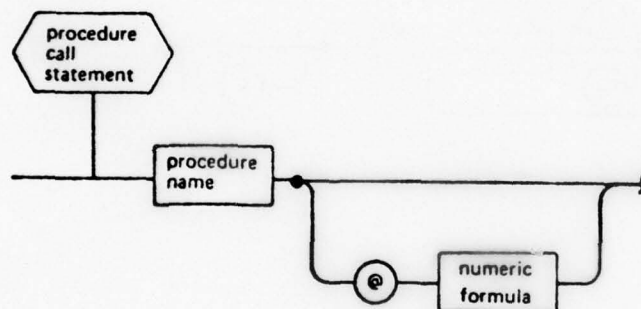
85: 31



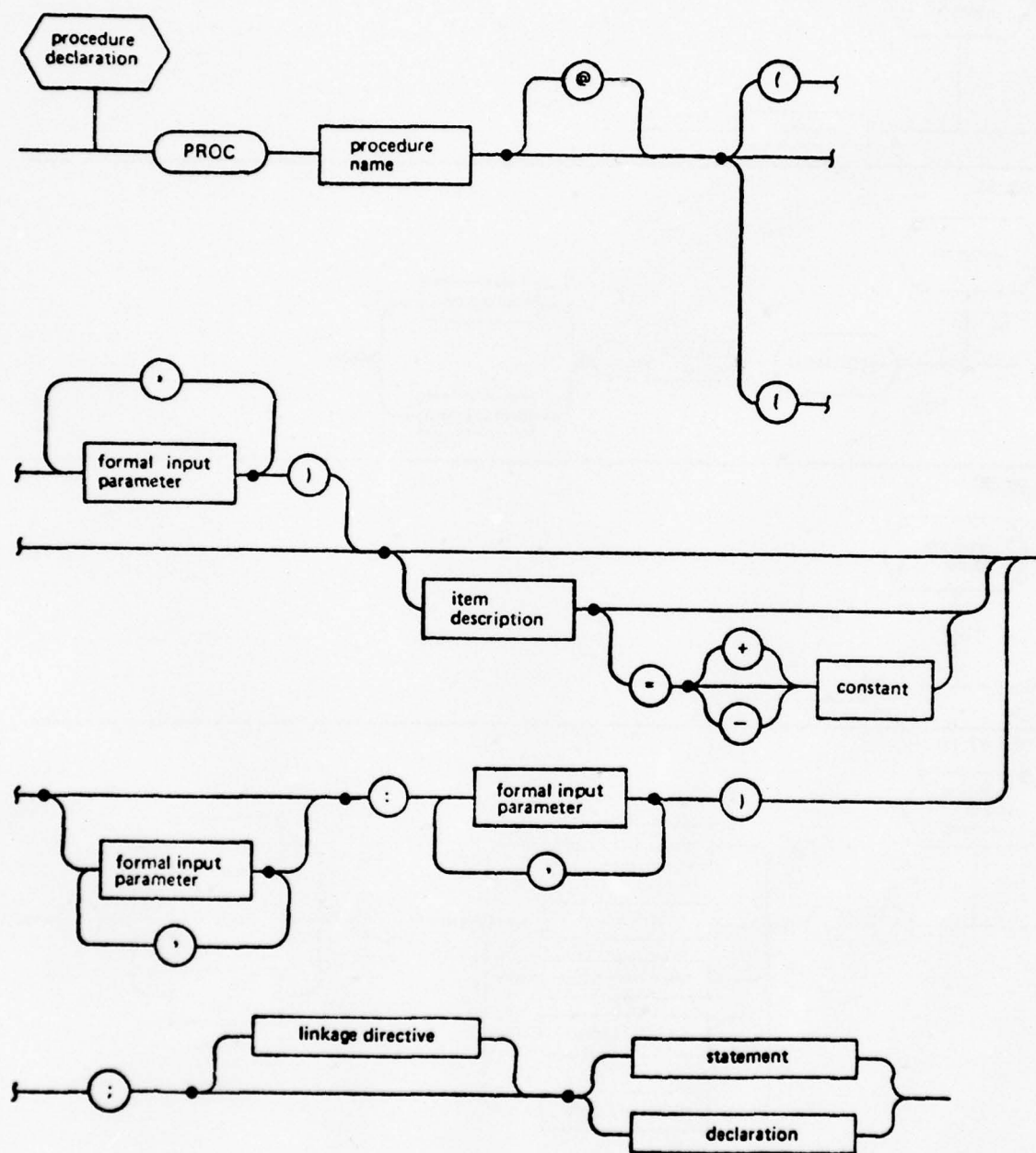
88: 87, 88



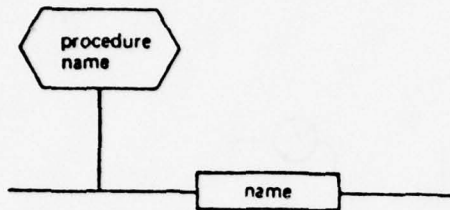
89: 98



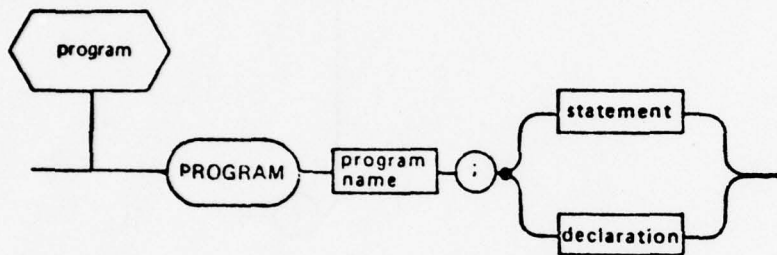
90: 23, 43, 65



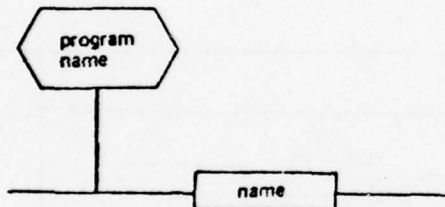
91: 2, 33, 51, 54, 70, 89, 90, 94, 95



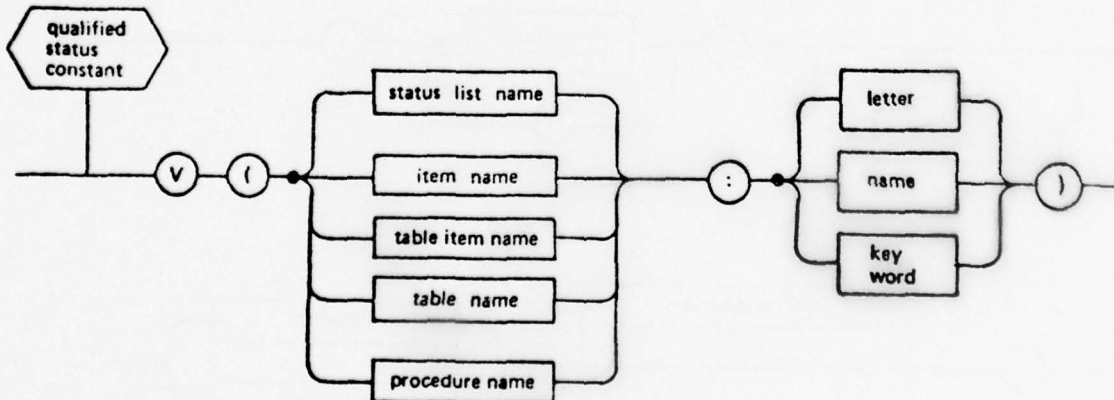
92: 65



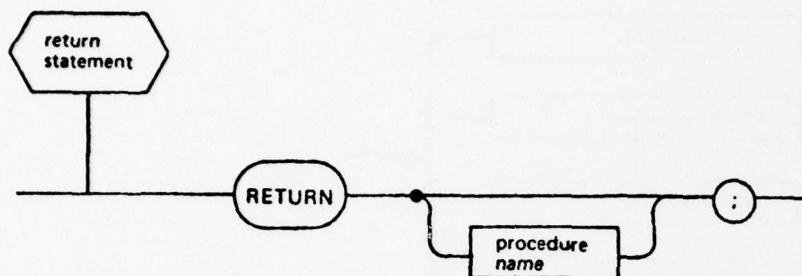
93: 92



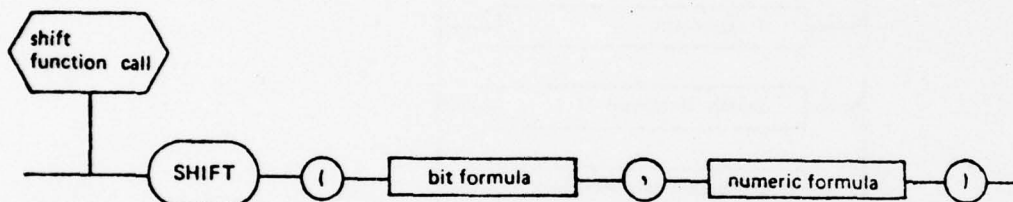
94: 57, 80



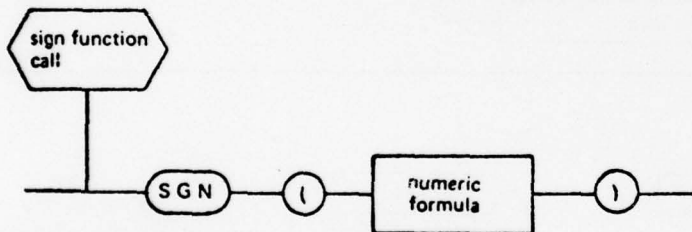
95: 98



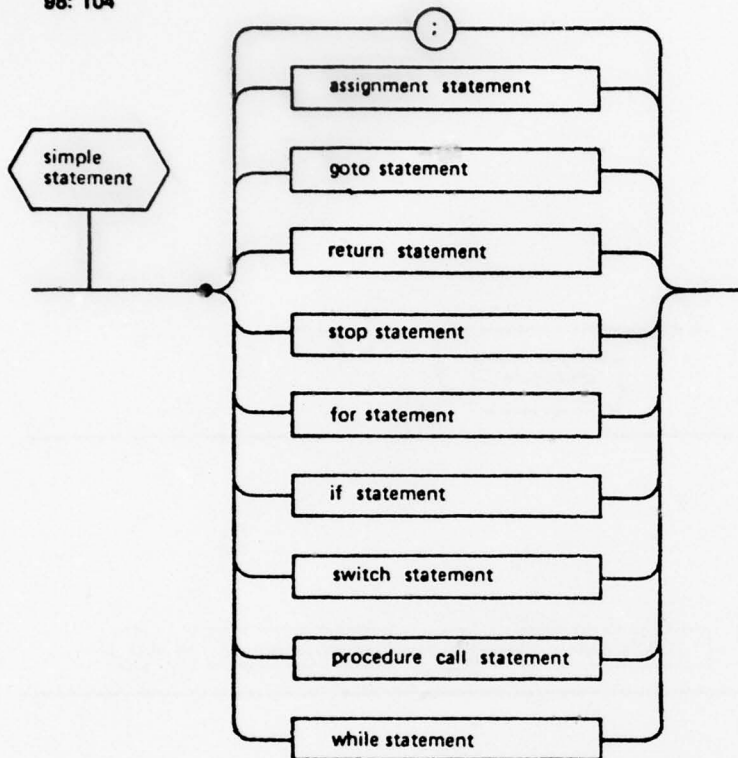
96: 7, 61



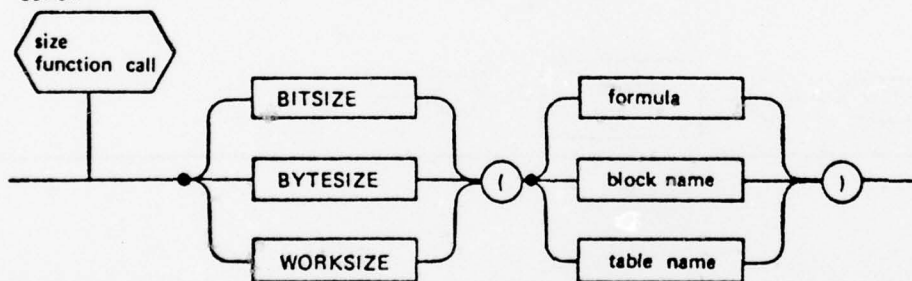
97: 61



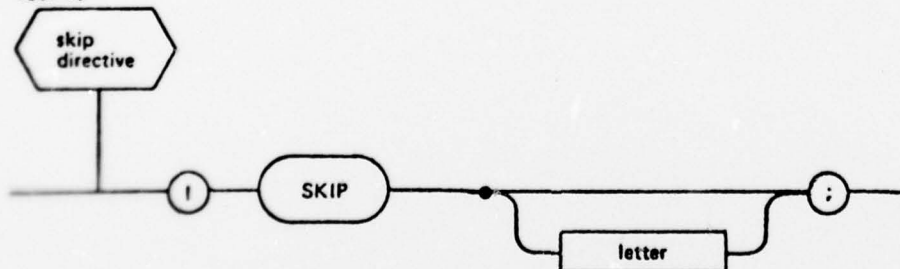
98: 104



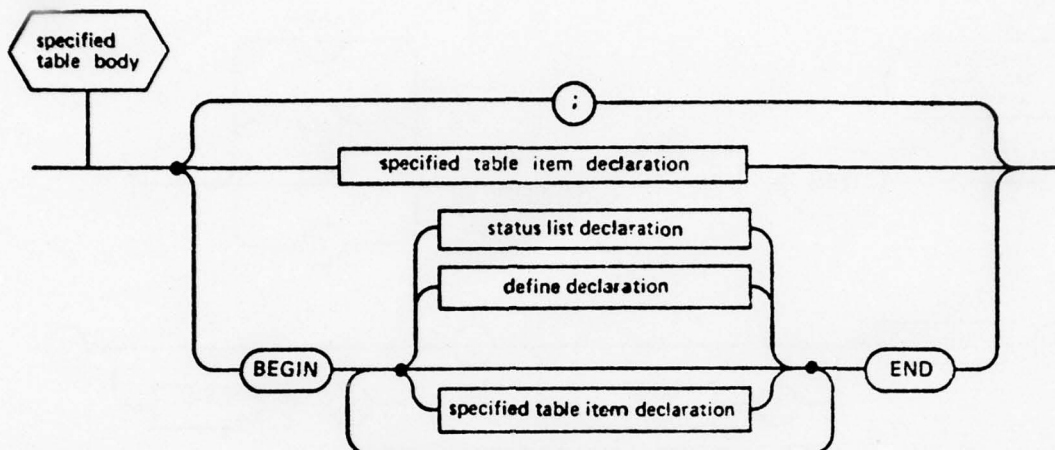
99: 61



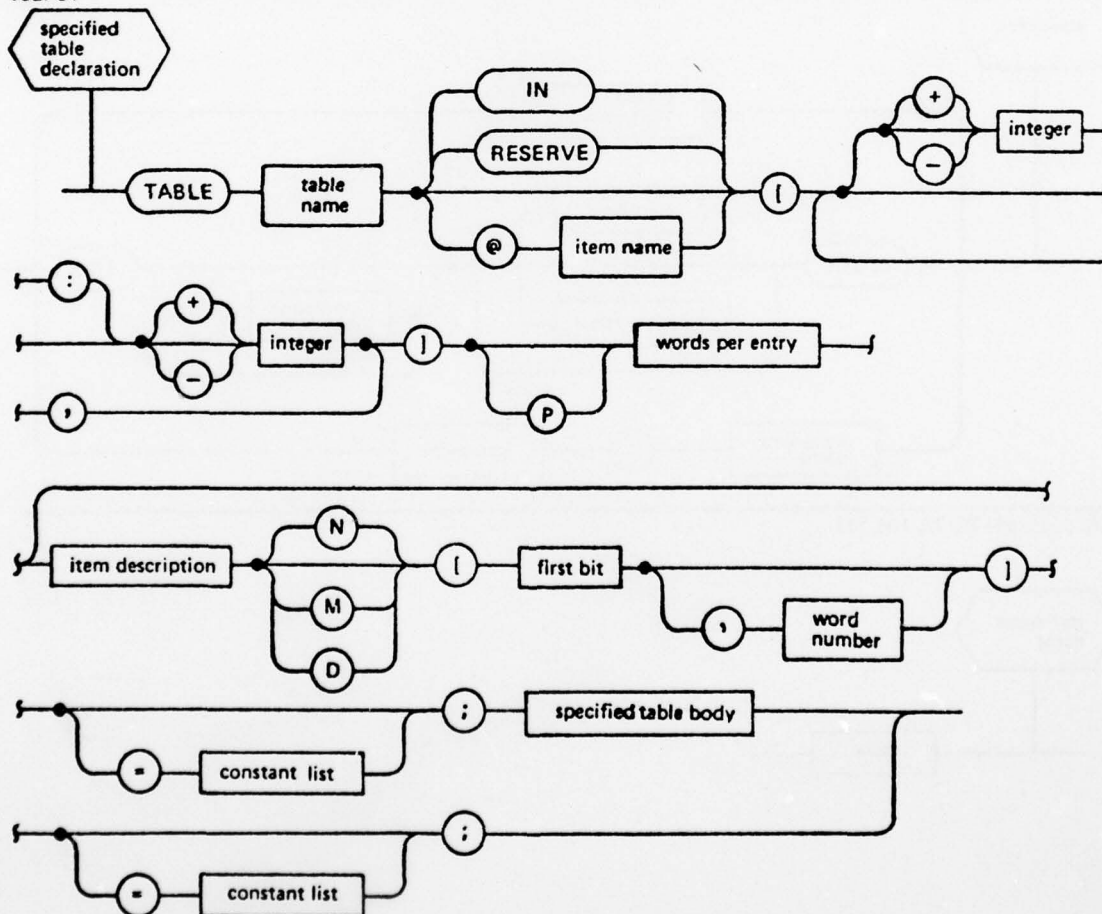
100: 40



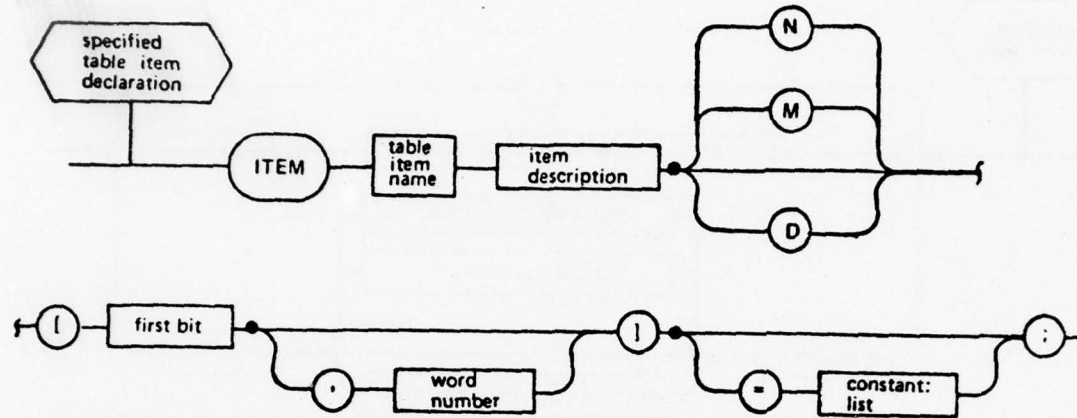
101: 102



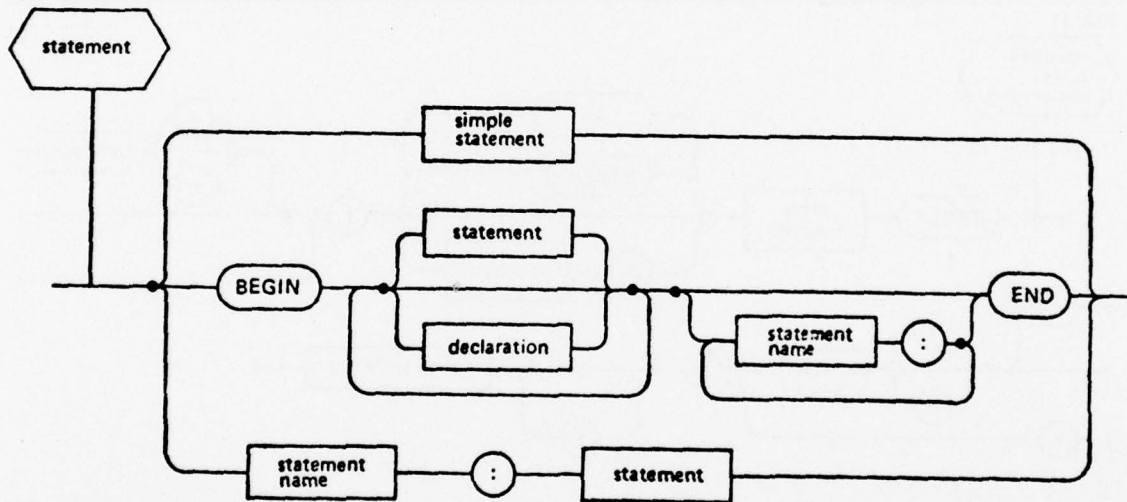
102: 31



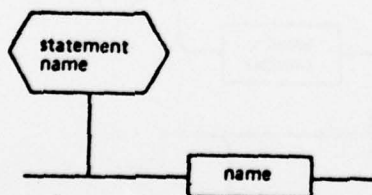
103: 101



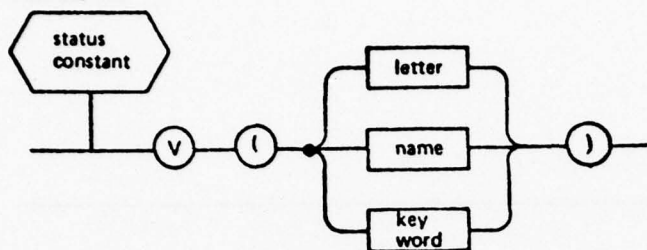
104: 50, 56, 90, 92, 104, 111, 116



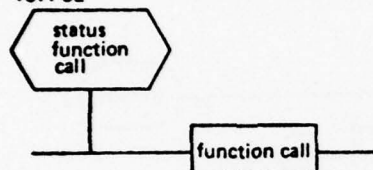
105: 2, 51, 55, 70, 73, 104, 111



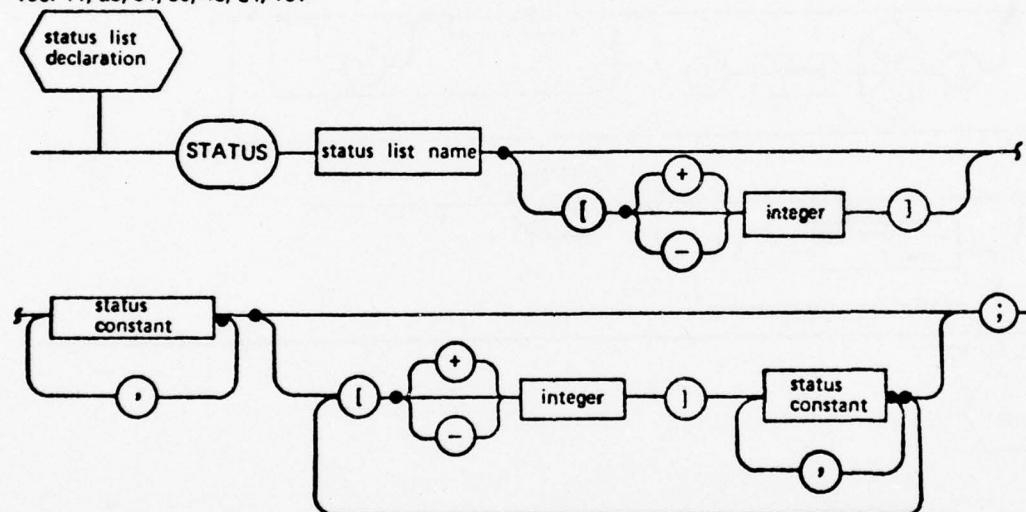
106: 80, 108



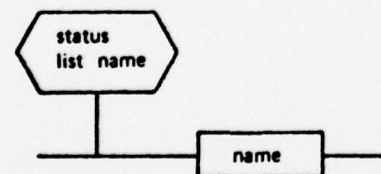
107: 82



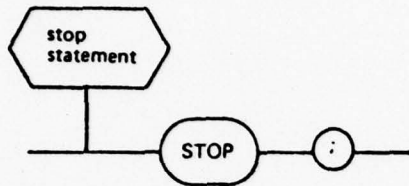
108: 11, 23, 34, 35, 43, 84, 101



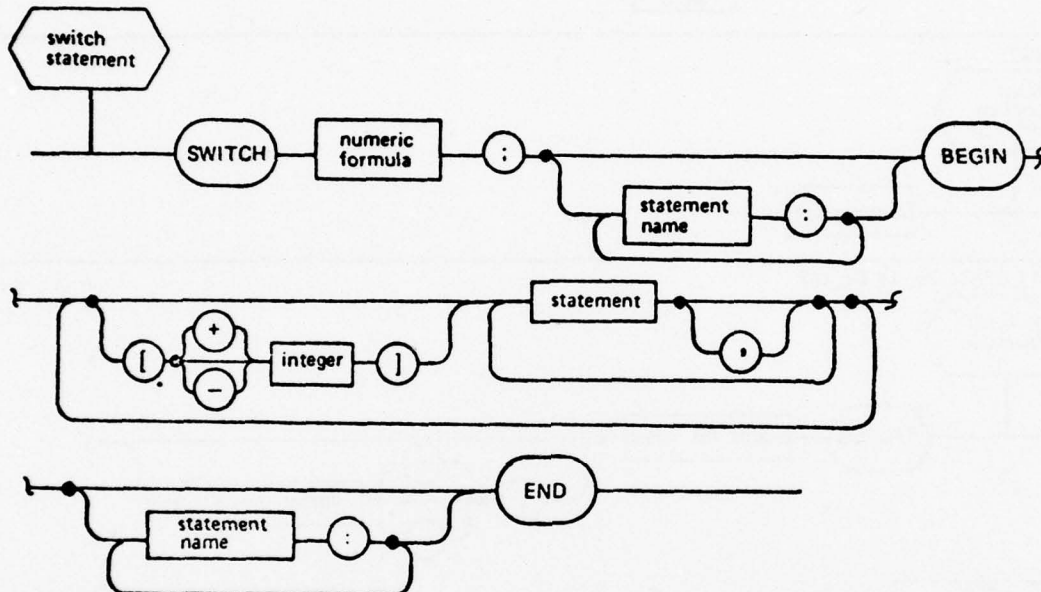
109: 63, 94, 108



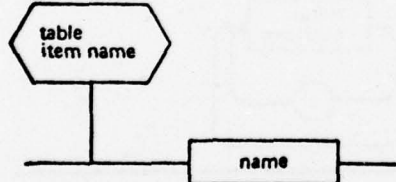
110: 98



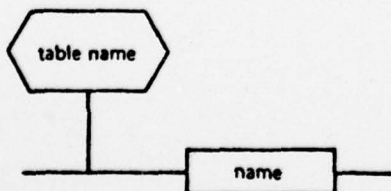
111: 98



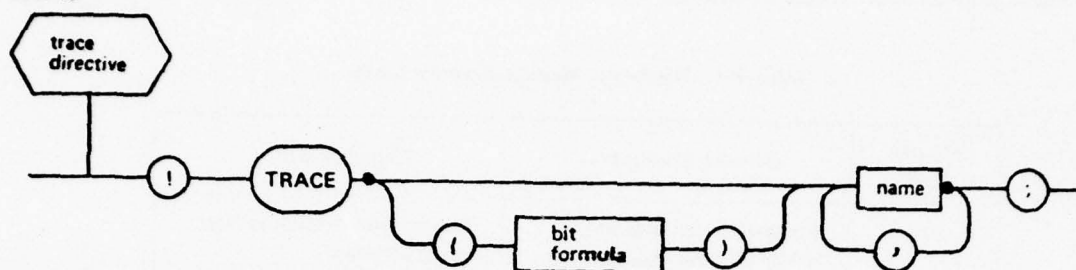
112: 74, 86, 94, 103



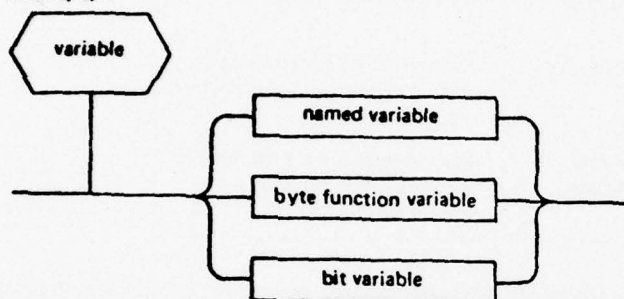
113: 2, 10, 32, 70, 74, 85, 94, 99, 102



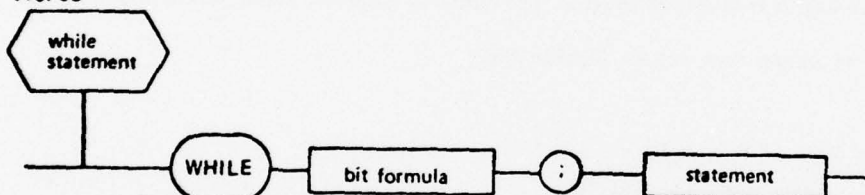
114: 40



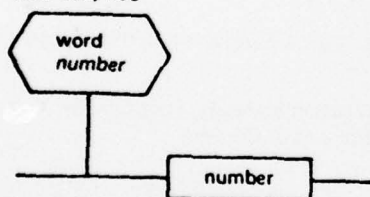
115: 2, 3, 4



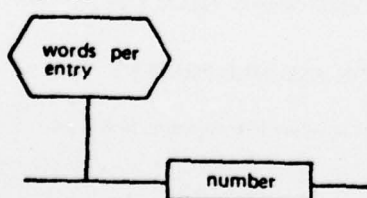
116: 98



117: 102, 103



118: 102



APPENDIX B. COMPILER ERROR MESSAGES

This appendix describes the compiler diagnostic messages which provide information about the source program and describe source program errors. Diagnostic messages are categorized according to severity level as described in the table below.

Table B-1. Diagnostic Message Severity Levels

<i>Severity Code</i>	<i>Severity Description</i>	<i>Compiler Action</i>
I	Information message only — no source error exists.	The compiler continues after listing message.
W	Possible error or actual error which compiler attempts to correct.	The compiler continues after taking corrective action.
S	Serious error which compiler cannot correct.	The statement containing the error is deleted and compilation is continued.
F	Fatal error which prevents completion of compilation.	The compilation is terminated abruptly. No relocatable program is produced.

Most source diagnostics are printed interspersed in the source listing immediately preceding the statement in error. Some diagnostics are printed at the end of the source listing. There is provision for inserting source program names at the beginning and end of the message text describing an error. The format of diagnostic messages is as follows:

**** !nnnn (ssss, cc) ** Message Text**

where:

I Severity level — I, W, S, or F.
nnnn Message number, defined in Table B-2.
ssss Number of statement to which the message refers. This is omitted when the message is not related to a statement.
cc Column position of the source to which the message refers. This is omitted when the column position is not known.
Message Text Text for source error or information which describes the situation in words. It optionally includes source program name inserts at the beginning and/or end of the text.

The following are examples of diagnostic messages:

**** S3028 (27, 10) ** TABLE1 HAS MORE DIMENSIONS THAN THE
 MAXIMUM LIMIT.**
**** W3019 (150, 47) ** MISSING SEMI-COLON ASSUMED PRESENT.**

Table B-2 presents the source diagnostic message text. Wherever an underline appears in a message, it will be replaced by a source program name.

Table B-2. Source Diagnostic Messages

<i>Message Number</i>	<i>Severity Code</i>	<i>Message Text</i>
1	F	INTERNAL COMPILER ERROR — ILLEGAL MESSAGE NUMBER
2	S	<u> </u> IS A MISSING OR ILLEGAL ITEM NAME
3	S	<u> </u> HAS AN ILLEGAL ENV/ALLOC SPECIFIER
4	S	<u> </u> HAS AN ILLEGAL POINTER ITEM
5	W	MISSING RIGHT PARENTHESIS ASSUMED PRESENT

<i>Message Number</i>	<i>Severity Code</i>	<i>Message Text</i>
6	S	_ HAS AN IMPOSSIBLE EXTRAD SPECIFICATION
7	S	_ HAS AN ILLEGAL INTEGER CONSTANT
8	S	_ HAS AN ILLEGAL INTEGER CONSTANT EXPRESSION
9	S	_ HAS A MISSING TYPE CODE
10	S	_ HAS AN ILLEGAL TYPE CODE
11	S	_ HAS AN IMPOSSIBLE SIZE SPECIFICATION
12	W	_ EXCEEDS MAX BYTE SIZE ALLOWED FOR CHAR VARIABLE
13	W	_ EXCEEDS MAXIMUM SIGNIFICAND SIZE
14	S	_ IS MISSING AN EXTRAD SPECIFICATION
15	W	_ EXCEEDS MAXIMUM EXPONENT SIZE
16	W	_ EXCEEDS MAXIMUM SIZE SPECIFICATION
17	S	_ HAS AN ILLEGAL STATUS LIST NAME
18	W	_ HAS A MISSING OR ILLEGAL PRESET CONSTANT
19	W	MISSING SEMI-COLON ASSUMED PRESENT
20	W	_ IS AN UNQUALIFIED STATUS CONSTANT
21	W	_ IS ILLEGALLY PRESET BECAUSE NOT IN RESERVE, PRESETS IGNORED
22	W	_ IS AN ILLEGALLY QUALIFIED STATUS CONSTANT
23	S	_ IS A MISSING OR ILLEGAL TABLE NAME
24	S	_ IS MISSING A DIMENSION LIST
25	W	_ HAS STATUS VALUES TOO LARGE FOR SIZE SPECIFICATION
26	S	_ HAS AN ILLEGAL DIMENSION LIST
27	S	_ HAS AN ILLEGAL RANGE OF VALUES IN DIMENSION LIST
28	S	_ HAS MORE DIMENSIONS THAN THE MAXIMUM LIMIT
29	W	MISSING RIGHT BRACKET ASSUMED PRESENT
30	W	_ EXCEEDS MAXIMUM WORDS/ENTRY SPECIFICATION
31	S	_ HAS AN ILLEGAL PACKING SPECIFIER
32	S	_ HAS AN IMPOSSIBLE WORDS/ENTRY SPECIFICATION
33	S	_ HAS AN IMPOSSIBLE ENTRY WORD SPECIFICATION
34	W	_ HAS A MISSING 'END' AFTER TABLE DECLARATION
35	S	_ IS MISSING A LEGAL FIRST BIT
36	W	_ SHOULD HAVE A ZERO FIRST BIT
37	S	_ SPECIFIES AN IMPOSSIBLE FIRST BIT
38	S	_ MUST BE DECLARED WITH ITS FIRST BIT ON A BYTE BOUNDARY
39	S	_ HAS A MISSING OR ILLEGAL ENTRY WORD POSITION
40	W	_ HAS AN ENTRY WORD POSITION OUT OF RANGE OF THE TABLE
41	S	_ IS A MISSING OR ILLEGAL STATUS LIST NAME
42	S	_ HAS A MISSING OR ILLEGAL INDEX IN STATUS LIST
43	S	_ HAS AN INDEX OUT OF RANGE FOR STATUS CONSTANTS
44	S	_ HAS A MISSING OR ILLEGAL STATUS CONSTANT
45	S	_ IS A MISSING OR ILLEGAL DEFINE NAME
46	W	_ HAS A DUPLICATE DEFINE PARAMETER
47	W	_ HAS A MISSING DEFINE PARAMETER
48	S	_ HAS A MISSING OR ILLEGAL DEFINE STRING
49	S	_ IS A MISSING OR ILLEGAL BLOCK NAME
50	W	MALFORMED STATUS CONSTANT
51	F	***UNUSED***
52	W	MISSING RIGHT PARENTHESIS
53	W	TOO MANY ACTUAL DEFINE PARAMETERS

<i>Message Number</i>	<i>Severity Code</i>	<i>Message Text</i>
54	W	ILLEGAL FORMAL DEFINE PARAMETER REFERENCE
55	W	ILLEGAL OCCURRENCE OF SEMI-COLON IN COMMENT
56	W	UNEXPECTED END-OF-FILE
57	F	***UNUSED***
58	F	***UNUSED***
59	W	MALFORMED PATTERN CONSTANT
60	W	ILLEGAL PATTERN DIGIT
61	W	ILLEGAL CHARACTER
62	W	COMMA OR RIGHT PAREN DOES NOT FOLLOW QUOTED ACTUAL DEFINE PARAMETER
63	W	RECURSIVE/CIRCULAR DEFINE USE IGNORED
64	W	ILLEGAL TOKEN AT STATEMENT START
65	W	UNIMPLEMENTED FEATURE
66	S	ILLEGAL OPERAND
67	F	STATEMENT STACK OVERFLOW
68	W	EXTRANEIOUS OR IMPROPERLY NESTED 'END'
69	W	_ :UNDEFINED VARIABLE
70	W	ILLEGAL 'BEGIN' IN EXTERNAL STRUCTURE
71	W	ILLEGAL REF/DEF IN EXTERNAL STRUCTURE
72	W	ILLEGAL REF/DEF
73	W	NAME/LITERAL DOES NOT FOLLOW '!COPY'
74	W	UNIMPLEMENTED DIRECTIVE
75	W	_ TABLE-ITEM NOT SUBSCRIPTED
76	W	UNSUBSCRIPTED TABLE NOT IN LOC OR PARM CONTEXT
77	W	MISSING LEFT PARENTHESIS
78	W	ILLEGAL NAME IN EXPRESSION
79	W	_ FUNCTION USED AS PROCEDURE OR VICE VERSA
80	F	ILLEGAL STACK OPERATOR
81	W	TOO MANY INDEX COMPONENTS
82	W	MISSING RIGHT SUBSCRIPT BRACKET
83	W	TOO FEW INDEX COMPONENTS
84	F	NO IL BUFFERS AVAILABLE
85	F	TRIAD TABLE OVERFLOW
86	F	NO TEMP SPACE AVAILABLE
87	W	INCOMPLETE EXPRESSION/MISSING OPERAND
88	F	EXPRESSION/OPERAND STACK OVERFLOW
89	F	EXPRESSION/OPERAND STACK UNDERFLOW
90	W	MISSING/ILLEGAL LOOP VARIABLE
91	W	LOOP VARIABLE UNDEFINED
92	W	MISSING FOR-COLON
93	W	MISSING SEMI-COLON
94	W	LHS CANNOT BE ASSIGNED-TO
95	W	RHS NOT A VALUE
96	S	ILLEGAL EXPRESSION OPERAND
97	S	MISSING POINTER EXPRESSION
98	S	MISSING COMMA/RIGHT PAREN
99	S	BYTE APPLIED TO NON-CHARACTER
100	W	'!LINKAGE' NOT WITHIN A PROC
101	S	_ IS A MISSING OR ILLEGAL PROC NAME

Message Number	Severity Code	Message Text
102	F	SCOPE TABLE OVERFLOW
103	W	INDEX COMPONENT LARGER THAN ONE CHARACTER, FIRST CHARACTER TAKEN
104	W	OUTERSCOPE PROC HAS '@', THIS ONE IS IGNORED
105	W	_IS NOT LEFT ADJUSTED IN FIELD - PACKING SPECIFICATION IGNORED
106	W	_IS NOT RIGHT ADJUSTED IN FIELD - PACKING SPECIFICATION IGNORED
107	W	_HAS DUPLICATE INPUT PARAMETERS
108	S	_HAS GARBAGE IN PARAMETER LIST
109	W	EXTRA COLON IN PARAMETER LIST IGNORED
110	S	_IS AN ILLEGAL FUNCTION SINCE IT HAS OUTPUT PARAMETERS
111	W	_IS AN ILLEGAL PROGRAM - CHANGED TO A PROC
112	W	MISSING SEMI-COLON OR RIGHT PARENTHESIS
113	F	INTERNAL COMPILER ERROR - OTPK
114	S	MISSING OR ILLEGAL NAME IN NAME DECLARATION
115	W	_HAS AN ILLEGAL ENV/ALLOC SPECIFIER, SPECIFIER IGNORED
116	W	_HAS AN ILLEGAL PRESET CONSTANT
117	W	UNIMPLEMENTED LINKAGE DIRECTIVE
118	W	PRESET INDEX COMPONENT OUT OF RANGE OF TABLE, PRESETS IGNORED
119	W	TOO MANY INDEX COMPONENTS IN PRESET, PRESETS IGNORED
120	W	ILLEGAL REPEAT COUNT, PRESETS IGNORED
121	W	EXTRA RIGHT PAREN IGNORED
122	W	PRESETS OUT OF RANGE OF TABLE, PRESETS IGNORED
123	W	TOO FEW INDEX COMPONENTS IN PRESET, PRESETS IGNORED
124	W	ILLEGAL PRESET INDEX COMPONENT, PRESETS IGNORED
125	W	_HAS AN ILLEGAL REF/DEF INSIDE A STRUCTURE, REF/DEF IGNORED
126	W	_HAS AN ILLEGAL REF/DEF BECAUSE IT IS A PARAMETER, REF/DEF IGNORED
127	W	ILLEGAL DIRECTIVE
128	W	DEFAULT ENV/ALLOC SPECIFICATION IGNORED
129	W	_HAS A SIZE SPECIFICATION TOO LARGE FOR MEDIUM PACKING, CHANGED TO DENSE
130	W	_HAS AN ILLEGAL REF/DEF BECAUSE NOT IN RESERVE
131	W	_IS AN ILLEGAL OUTPUT PARAMETER
132	W	_IS NOT A PARAMETER IN A REF PROC
133	W	ILLEGAL 'REF' IN COMPOOL ASSEMBLY, 'REF' IGNORED
134	F	MISSING OR ILLEGAL COMPOOL NAME
135	F	MISSING COMPOOL STATEMENT
136	F	ILLEGAL COMPOOL STATEMENT
137	W	COMPOOL NAME/LITERAL DOES NOT FOLLOW '!COMPOOL', DIRECTIVE IGNORED
138	W	COMPOOL NAME TABLE OVERFLOW, REST OF DIRECTIVE IGNORED
139	W	MISSING OR ILLEGAL NAME ON COMPOOL DIRECTIVE
140	W	NOT FOUND IN COMPOOL, NAME IGNORED
141	W	'!COMPOOL' NOT ALLOWED IN A COMPOOL ASSEMBLY
142	W	ILLEGAL COMPOOL DIRECTIVE
143	W	ILLEGAL '!SKIP' BETWEEN '!BEGIN' AND '!END'

<i>Message Number</i>	<i>Severity Code</i>	<i>Message Text</i>
144	F	TOO MANY CONDITIONAL LEVELS
145	W	NESTED BEGIN WITH SAME OR NO LETTER
146	W	EXTRANEIOUS OR IMPROPERLY NESTED !END
147	W	NO 'END' FOR 'BEGIN' AT STNO:
148	W	_ HAS AN ILLEGAL TRACE CLASS
149	W	_ IS AN ILLEGAL 'LIST' PARAMETER
150	S	MISSING COMMA
151	S	DSIZE APPLIED TO NON-BASED PROC
152	S	DSIZE APPLIED TO NON-PROC
153	W	COPY LEVEL EXCEEDED
154	S	DECL ERROR PREVENTS EXPR ANALYSIS
155	S	LOC OPERAND MUST BE LABEL, PROC, TABLE, BLOCK, OR NAMED VARIABLE
156	W	ILLEGAL PATTERN CONSTANT BASE
157	W	'0.0' ASSUMED FOR '.'
158	S	INCOMPLETE PROGRAM/COMPOOL:
159	S	INCOMPLETE 'IF/ELSE' AT STNO:
160	S	INCOMPLETE 'FOR' AT STNO:
161	S	NO 'END' FOR 'BEGIN' AT STNO:
162	S	INCOMPLETE PROC AT STNO:
163	S	INCOMPLETE BLOCK AT STNO:
164	W	-PARAMETER LABEL CANNOT BE DEFINED
165	W	'GOTO' ASSUMED PRECEDING-
166	S	MISSING NAME
167	S	UNDEFINED LABEL:
168	W	MISSING PROC/PROG/COMPOOL HEADER
169	W	ILLEGAL LINKAGE DIRECTIVE
170	W	ILLEGAL PHASE NUMBER IN ON/OFF DIRECTIVE
171	W	ILLEGAL BIT NUMBER IN ON/OFF DIRECTIVE
172	S	BASED-PROC CALL HAS MISSING BASE
173	S	MISSING ACTUAL PARAMETERS
174	S	MISSING PARAMETER DELIMITER
175	S	TOO MANY ACTUAL PARAMETERS FOR PROC:
176	S	INPUT/OUTPUT PARAMETER MISMATCH FOR PROC:
177	S	UNDECLARED FORMAL PARAMETERS
178	S	FORMAL LABEL PARAMETER MUST BE OPPOSITE ACTUAL LABEL
179	S	FORMAL ITEM PARAMETER MUST BE OPPOSITE ACTUAL FORMULA
180	S	FORMAL PROC PARAMETER MUST BE OPPOSITE ACTUAL PROC
181	S	OUTPUT PARAMETER IS NOT A VARIABLE
182	W	_ IS A PART-WORD OR NON-CONTIGUOUS MULTI-WORD OPPOSITE A NAME PARAMETER
183	S	ACTUAL PARAMETER MUST BE A TABLE, BLOCK, NAMED VARIABLE, CONSTANT, OR @-FORMULA
184	W	TOO MANY COLONS
185	S	TOO FEW ACTUAL PARAMETERS
186	W	RETURN REFERENCES NON-ACTIVE PROC:
187	S	UNDEFINED NAME:
188	I	CONVERSION REQUIRED
189	F	TRACE TABLE OVERFLOW

<i>Message Number</i>	<i>Severity Code</i>	<i>Message Text</i>
190	F	PROC STACK OVERFLOW - CIP
191	W	DUPLICATE NAME
192	F	BAD SRCH PARAMETER(S)
193	S	MISSING OR ILLEGAL TRACE NAME
194	W	_ IS AN ILLEGAL TRACE PROC
195	I	: UNDECLARED INPUT PARAMETER-DEFAULTS TO LABEL
196	S	: UNDECLARED OUTPUT PARAMETER
197	S	MISSING OR ILLEGAL SWITCH INDEX
198	W	DUPLICATE SWITCH POINT INDEX
199	S	INCOMPLETE 'SWITCH' AT STNO:
200	S	INCOMPLETE 'COMPOOL' AT STNO:
201	S	MISSING OR ILLEGAL ADDRESS IN OVERLAY STATEMENT
202	S	OVERLAY ADDRESS OUT OF RANGE
203	S	_ IS A MISSING OR ILLEGAL NAME IN OVERLAY STATEMENT
204	S	_ IS AN ILLEGAL BASED NAME IN OVERLAY STATEMENT
205	S	_ IS AN ILLEGAL EXTERNAL NAME IN OVERLAY STATEMENT
206	S	OVERLAY NAMES NOT IN SAME STRUCTURE
207	S	_ IS AN ILLEGAL NAME IN OVERLAY WITH ADDRESS SPECIFICATION
208	S	OVERLAY STACK OVERFLOW

APPENDIX C. COMPILATION AND EXECUTION OF J73/I PROGRAMS

COMPILATION

In order to execute a J73/I source program, the program must first be translated from source to relocatable form using the J73/I compiler. The compiler is executed with the DEC-10 Operating System command:

R J73I

This command is followed by one or more lines of compilation parameters. Each parameter line completely describes a compilation, thus allowing multiple compilations with a single execution of the compiler. For compilations initiated from a terminal, the compiler prompts for each parameter line with an asterisk (*). For batch job initiated compilations, the first non-parameter line must contain a period as the first character.

The J73/I compiler parameter syntax is similar to other DEC-10 language processor's parameter syntax. The general syntax of a parameter line is as follows:

* relfile, listfile=sourcefile, compoolfile

where

relfile	Identifies the relocatable file to be produced. The general syntax of a file identification is device type: filename.extension If a relocatable file is to be produced by the compiler, at least one of these three file identification fields must be specified. If the device type is not specified, DSK is used. If extension is not specified, .REL is used. If filename is not specified, the source file name is used.
listfile	Identifies the list file. If absent, no compiler list output will be provided. If the device is not specified, DSK is used. If the filename is not specified, the source file name is used. If extension is not specified, .LST is used.
sourcefile	Identifies the J73/I source file to be compiled. This parameter must be present. If the device type is not specified, DSK is used.
compoolfile	This field, when present, identifies a special mode of compilation called a "compool build" and defines the name of the compool dictionary file to be produced by the compiler. This compool dictionary file will be accessed in subsequent compilations using the !COMPOOL directive. If the device type is not specified, DSK is used. If the file name is not specified, the source file name is used. If extension is not specified, .CMP is used.

Compilation options are selected with switches imbedded in a parameter line. The syntax of a switch is a slash (/) followed by the switch name. The position of a switch within a parameter line has no significance. A switch may appear following a file specification or another switch.

The following switches are defined for the J73/I compiler, where the underlined letters are the minimum abbreviation allowed for the switch.

<u>Switch Name</u>	<u>Meaning</u>
<u>C</u> CROSSREF	Generate cross reference listing including only referenced names.
<u>A</u> CROSS	Generate cross reference listing including all names (both referenced and unreferenced).
<u>S</u> YNTAX	Perform only syntax checking.
<u>I</u> NDENT	Produce indented (formatted) source listing.
<u>H</u> BC	Generate code for the Hot Bench Computer. In the absence of this switch, code is produced for the DEC-10.
<u>N</u> OSTAT	Suppress statistics summary listing.
<u>M</u> ACROCODE	Generate object code (macrocode) listing.
<u>L</u> OWSEG	Generate data in low segment. If no HIGHSEG switch, also generate code in low segment.
<u>H</u> IGHSEG	Generate code in high segment. If no LOWSEG switch, also generate data in high segment. If both LOWSEG and HIGHSEG are selected or neither are selected, data will be generated in the low segment and code will be generated in the high segment.

<u>NONEST</u>	This switch disallows nested procedures. When the switch is selected and the source program contains nested procedures, a fatal diagnostic message is issued.
<u>ISD</u>	Produce internal symbol dictionary. This is data used by SDVS which describes the statements and variables in the program.
<u>NOPT</u>	Do not perform optimization.
<u>LISTCOPY</u>	List files included with the !COPY directive. In the absence of this switch, these files are not listed.
<u>NOSOURCE</u>	Do not produce a source listing. This switch is useful for getting diagnostic messages on the list file without also getting the source program listing.
<u>DEFINE</u>	This switch is only effective when the INDENT switch is also present. It causes characters which are substituted in the source program for a DEFINE call to appear in the source listing in place of the DEFINE call.
<u>NOTRACE</u>	This switch causes the compiler to suppress the generation of code which is produced because of the TRACE directive. Since trace output is normally used for debugging purposes, the NOTRACE switch allows this debugging code to be deleted from the relocatable program without deleting it from the source program.
<u>NOINFORM</u>	This switch suppresses the listing of information level messages.
<u>NOWARNING</u>	This switch suppresses the listing of both warning and information level messages.

The following are examples of DEC-10 commands to perform J73/I compilation.

```
.R J73I
*,LPT:=REDUCE/SYNTAX/INDENT
```

The example above compiles source program REDUCE. It performs only syntax checking (i.e., no relocatable file is produced). The source listing is indented and directed to the line printer.

```
.R J73I
*PROG,PROG=PROG/C
*PROC1,PROC1=PROC1/C
*PROC2,PROC2=PROC2/C
```

The example above illustrates compilation of three programs with a single invocation of the compiler. PROG, PROC1, and PROC2 are each compiled with a cross reference listing. The outputs are PROG.LST, PROC1.LST, PROC2.LST, PROG.REL, PROC1.REL, and PROC2.REL.

```
.R J73I
*VEL,TTY:=VELOC/M/A/NOIN/H
```

The example above compiles source program VELOC, produces relocatable file VEL.REL and directs list output to the terminal. Listing options selected are object code listing (macrocode), cross reference listing containing all names (both referenced and unreferenced) and suppression of printing of information level messages. Both code and data will be compiled into the high segment.

DEC-10 EXECUTION

The output of a J73/I compilation is a relocatable module file with extension .REL. In order to execute a program on the DEC-10, it must first be converted from relocatable to core image form using the LINK-10 linking loader. For a complete description of LINK-10, refer to the LINK-10 Programmer's Reference Manual, no. DEC-10-ULKMA-B-D. The following paragraphs describe a subset of facilities which allow simple usage of LINK-10 for common applications involving non-overlaid programs.

LINK-10 is invoked with the command:

```
.R LINK
```

This is followed by one or more lines of link parameters. For terminal usage of LINK-10, for each line of link parameters the user is prompted with an asterisk.

Parameter lines contain file specifications, separated by commas, and imbedded LINK-10 option switches.

AD-A044 915

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 3/1
A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CON--ETC(U)
MAY 77 G E SCHWEIZER, A A CALLAWAY, E C GANGL

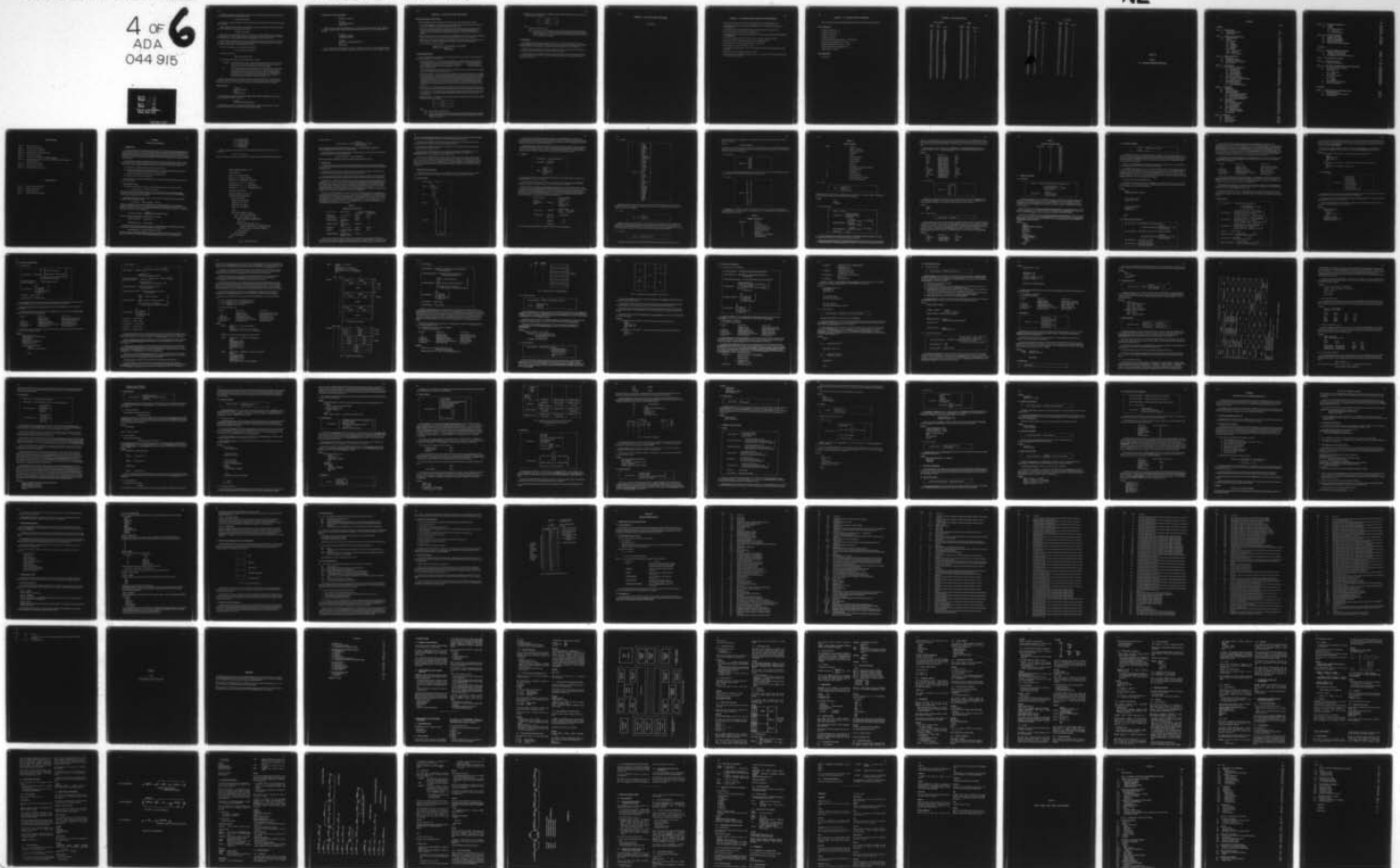
UNCLASSIFIED

AGARD-AR-90

NL

4 OF
ADA
044 915

6



File specifications may identify either input or output files. An input file specification identifies a relocatable file to be included in the program. It has the following form:

device:filename.extension

If devicetype: is not present, DSK: is assumed. If extension is present, it must be .REL. If extension is not present, .REL is assumed.

Output file specifications are optional. They identify the core image to be produced and the memory map listing file. The core image file is specified using the following form:

devicetype:filename/SAVE

If devicetype: is not specified, DSK: is assumed. If filename is not specified, the file name of the main program relocatable file is used. If the core image contains high and low segments, two files are created with extensions .HGH and .LOW, respectively. If the core image contains only a single segment, the file extension is .SAV.

If /SAVE is not used, LINK-10 does not create a file containing the core image. After execution of LINK-10, the core image is "loaded" and may be directly executed with the RUN command with no file specification. The loaded core image may also be copied to a file using the SAVE command.

The memory map listing file is specified using the following form:

devicetype:filename.extension/MAP

The default specification is:

DSK:nameofmainprogram.MAP

Switches indicate LINK-10 options. The following switches are defined.

- | | |
|----------|---|
| /EXECUTE | The /EXECUTE switch is used to specify that the loaded program is to be started at its entry point (i.e., the start address) upon completion of loading. This switch does not cause the termination of loading; the /GO switch is needed to terminate loading. |
| /GO | The /GO switch is used to terminate the loading process. When LINK-10 executes the /GO switch, it finishes loading the current specification, searches default libraries, produces the requested output files, and either exits to the monitor or runs the core image produced depending upon the switches appearing in the command strings. If the /EXECUTE switch has been specified, execution begins at the normal start address of the user's program. If /EXECUTE has not been specified, LINK-10 exits to the monitor. |

LINK-10 automatically includes a module which satisfies an unresolved external reference only if the module is contained in a library. All modules to be included in a link which are not in a library must be explicitly identified to LINK-10 on a parameter line. For a discussion of libraries, see the LINK-10 Programmer's Reference Manual.

LINK-10 Examples

```
.R LINK
*PROG1,PROG/SAVE
*PROC1
*PROC2,PROC3/GO
```

The example above combines relocatable modules PROG1, PROC1, PROC2, and PROC3 and creates the core image file named PROG. PROG is not executed.

```
.R LINK
*MAIN,SUB1,SUB2/EXECUTE/GO
```

The example above combines relocatable modules MAIN, SUB1, and SUB2, loads the resultant core image program without creating a file for it and transfers control to the main program.

Compilation, Link, and Execution Example

```
.R J73I
*SORT1,LPT:=SORT1/C
*
.R LINK
*SORT1,SORT/SAVE/GO
.RUN SORT
```

The example above compiles program SORT1 with a cross reference listing directed to the line printer. The relocatable SORT1 is then linked and core image files SORT.HGH and SORT.LOW are created. Finally, program SORT is executed.

```
.R J73I
*MAIN,MAIN:=MAIN/M
*SUB1,SUB1:=SUB1/M
*
.R LINK
*MAIN,SUB1,LPT:/MAP,PROG1/SAVE
*/GO
.RUN PROG1
```

In the example above, program MAIN and proc SUB1 are compiled with an object code listing. Relocatables MAIN and SUB1 are then linked to become PROG1. A memory map is directed to the line printer. Finally, PROG1 is executed.

APPENDIX D. J73/I DEC-10 RUN TIME CONVENTIONS

PROCEDURE LINKAGE CONVENTIONS

The standard DEC-10 linkage convention is used by J73 object code. The convention used is as follows:

- R17 describes a linkage stack. The right half contains the address-1 of the next free stack word. The left half contains the complement of the number of words-1 unused on the stack.
- R16 is used as a parameter list pointer. A parameter list is a series of parameter addresses right justified and stored one per word. The left half of each parameter list entry is zero. Preceding the parameter list is a parameter count word containing zero in the right half and the negative of the number of parameters in the left half.
- R0 is used as the function value return register for calls to FORTRAN functions. For J73 functions, an additional parameter (described by the last entry in the parameter list) is passed to receive the function result.
- Registers R0, R1 and R16 are considered to be volatile registers. That is, their contents need not be preserved by the called procedure. Registers R2 through R15 and R17 must be preserved by the called procedure.
- A call to a procedure P1 is done as follows:

```
MOVEI R16,PL ; GET PARAMETER LIST ADDRESS
PUSHJ R17,P1 ; CALL P1
...          ; RETURN HERE
```

J73 PARAMETER PASSING

J73 procedure parameters are passed using the standard DEC-10 parameter list convention. Each actual parameter is passed using a parameter list word as follows:

- Value input parameters – If a type conversion must be applied to the actual parameter or the actual parameter is not contained in storage exactly as the formal parameter (stored in consecutive full words) then the value of the actual parameter, converted if necessary, is assigned to a temp before the call. If this temp assignment is done, the address of the temp is passed in the parameter list. Otherwise, the address of the actual parameter is passed in the parameter list.

The called procedure prologue (the initial code for the procedure) copies value input parameters from the address specified in the parameter list to the formal parameter.

- Value output parameters – If a type conversion must be applied to the formal parameter before it can be assigned to the actual parameter or if the actual and formal parameter are not allocated to the same number of consecutive full words, the address of a temp which matches the formal parameter is passed in the parameter list. Otherwise, the actual parameter address is passed in the parameter list. The called procedure epilogue (the code executed immediately before procedure exit) copies each value output parameter to the address specified in the corresponding parameter list entry. Upon return from the procedure call, for each output value parameter for which a temp was passed, the calling procedure will copy the value contained in the temp to the corresponding actual parameter.
- Name parameters – For all name parameters, the address of the actual parameter is passed in the parameter list. The called procedure will use this actual parameter address for all access to the formal parameter in the procedure.
- J73 function results – J73 function results are returned using a compiler generated value output parameter as the final parameter. The calling procedure supplies in the parameter list the address of a temp which is the size of the function result.
- Parameter procedures – The address of a two word packet is passed in the parameter list for parameter procedures. The format of the packet is as follows:

PEP
PAP

where

PEP is the procedure entry point address

PAP is the procedure @ space pointer. This value will be placed in R15 immediately before calling the procedure. It will only be used by a procedure which is internal to an @ procedure to locate the procedure's local storage.

- Parameter labels – The address of a two word packet is passed in the parameter list for parameter labels. The format of the packet is as follows:

LADR
RPKT

where

LADR is the address of the label.

RPKT is the address of a two word register save packet which contains values to load in registers R17 and R15 respectively. It is only valid to transfer to a label parameter if the procedure containing the label is currently active. The values for R17 and R15 are those established for the registers in the containing procedure prologue.

@ PROCEDURES

@ procedures have their local storage supplied by their caller. In a call to an @ procedure, the value of the @ expression defining the called procedure's local storage is passed as the first parameter to the procedure. For example, the call P1@XX(YY,ZZ); has a parameter list which describes the parameters, XX, YY and ZZ respectively.

Within an @ procedure, register R15 is dedicated to containing the pointer to local storage. R15 is loaded in an @ procedure prologue, after saving the previous contents of R15 in local storage.

Procedures inner to an @ procedure have their local storage as part of the containing @ procedure's local storage. These procedures therefore also use R15 to access their local storage, although the value is contained in R15 upon entrance to the procedure.

APPENDIX E. J73/I HBC RUN TIME CONVENTIONS

[Not available.]

APPENDIX F. J73/I COMPILER LIMITS, CAPACITIES AND RESTRICTIONS

The following limits, capacities and restrictions are imposed by the J73/I compiler implementation.

- The maximum number of levels of nesting of PROGRAMs, PROCs, IFs, FORs, SWITCHs and BLOCKs is approximately 50.
- The maximum number of symbols within a language construct terminated by a semicolon is 100.
- The maximum number of procedures per source program is 127.
- The maximum number of levels of nesting of COPY files is 10. There is no limit on the total number of COPY or COMPOOL files.
- The maximum number of unique names allowed in a source program is approximately 10000.
- The maximum number of diagnostic messages allowed for a compilation is 500.
- The maximum size of a character variable or constant is 255 characters.
- The maximum size of a pattern constant is 1785 bits.
- The range of status values allowed is -511 to 511. The maximum number of status values allowed in a status list is 1023.
- The maximum number of dimensions allowed for a table is 7.
- The maximum number of significant characters in a name is 31. However, for the DEC-10 external names must be unique in the first 6 characters.

APPENDIX G. J73/I TARGET MACHINE PARAMETERS

This section describes target machine parameters which must be considered when writing J73/I programs.

DEC-10 PROGRAMS

- Number of bits per word – 36
- Number of bytes per word – 5
- Number of bits per byte – 7
- Address units per word – 1
- Maximum integer arithmetic operand size – 35 bits plus sign
- Maximum bit formula operand size – 1785 bits
- Maximum logical formula operand size – 36 bits
- Maximum number of words per table entry – 51
- Medium packing – half word.

HBC PARAMETERS

[Not available.]

APPENDIX H. ASCII CHARACTER SET

Control Characters			Figures		
<i>Decimal Value</i>	<i>Octal Value</i>	<i>Character</i>	<i>Decimal Value</i>	<i>Octal Value</i>	<i>Character</i>
000	000	NUL	032	040	space
001	001	SOH	033	041	!
002	002	STX	034	042	"
003	003	ETX	035	043	#
004	004	EOT	036	044	\$
005	005	ENQ	037	045	%
006	006	ACK	038	046	&
007	007	BEL	039	047	'
008	010	BS	040	050	(
009	011	HT	041	051)
010	012	LF	042	052	*
011	013	VT	043	053	+
012	014	FF	044	054	,
013	015	CR	045	055	-
014	016	SO	046	056	.
015	017	SI	047	057	/
016	020	DLE	048	060	0
017	021	DC1	049	061	1
018	022	DC2	050	062	2
019	023	DC3	051	063	3
020	024	DC4	052	064	4
021	025	NAK	053	065	5
022	026	SYN	054	066	6
023	027	ETB	055	067	7
024	030	CAN	056	070	8
025	031	EM	057	071	9
026	032	SUB	058	072	:
027	033	ESC	059	073	;
028	034	FS	060	074	<
029	035	GS	061	075	=
030	036	RS	062	076	>
031	037	US	063	077	?

Upper Case			Lower Case		
<i>Decimal Value</i>	<i>Octal Value</i>	<i>Character</i>	<i>Decimal Value</i>	<i>Octal Value</i>	<i>Character</i>
064	100	@	096	140	`
065	101	A	097	141	a
066	102	B	098	142	b
067	103	C	099	143	c
068	104	D	100	144	d
069	105	E	101	145	e
070	106	F	102	146	f
071	107	G	103	147	g
072	110	H	104	150	h
073	111	I	105	151	i
074	112	J	106	152	j
075	113	K	107	153	k
076	114	L	108	154	l
077	115	M	109	155	m
078	116	N	110	156	n
079	117	O	111	157	o
080	120	P	112	160	p
081	121	Q	113	161	q
082	122	R	114	162	r
083	123	S	115	163	s
084	124	T	116	164	t
085	125	U	117	165	u
086	126	V	118	166	v
087	127	W	119	167	w
088	130	X	120	170	x
089	131	Y	121	171	y
090	132	Z	122	172	z
091	133	[123	173	{
092	134	\	124	174	
093	135]	125	175	}
094	136	^	126	176	~
095	137	_	127	177	DEL

ANNEX H

JOVIAL

B-1 AVIONICS SUPPORT SOFTWARE

CONTENTS

	Page
CHAPTER I	
Section 1 — INTRODUCTION	H-5
1.1 The Descriptive Notation	H-5
1.2 Semantic Types	H-7
Section 2 — ELEMENTS OF THE LANGUAGE	H-8
2.1 Characters and Signs	H-8
2.2 Symbols	H-9
2.2.1 Primitives	H-10
2.2.2 Names	H-10
2.2.3 Comments	H-10
2.2.4 Abbreviations	H-11
2.2.5 Ideograms	H-11
2.2.6 Literals	H-12
2.2.6.1 Numeric Literals	H-12
2.2.6.2 Bit Literals	H-13
2.2.6.3 Character Literals	H-13
Section 3 — FORM OF A PROGRAM	H-14
3.1 General Form	H-14
3.2 Compile-Time COMPOOL	H-15
3.3 Procedure and Function Definitions	H-15
3.4 Block Definitions	H-16
Section 4 — DECLARATIONS	H-17
4.1 Data Definition and Organization	H-18
4.1.1 Item Declarations	H-18
4.1.2 Table Declarations	H-19
4.1.3 Array Declarations	H-22
4.1.4 Overlay Declarations	H-23
4.1.5 Block Declarations	H-23
4.2 Execution Control Declarations	H-25
4.2.1 Procedure and Function Declarations	H-25
4.2.2 Switch Declarations	H-26
4.3 Name-Defining Declarations	H-27
4.3.1 Define Declarations	H-27
4.3.2 Constant Name Declarations	H-27
Section 5 — STATEMENTS	H-28
5.1 Null Statement	H-28
5.2 Compound Statements	H-29
5.3 Assignment Statements	H-29
5.3.1 Numeric Assignment Statements	H-29
5.3.2 Bit Assignment Statements	H-31
5.3.3 Character Assignment Statements	H-31
5.3.4 Entry Assignment Statements	H-31
5.4 Call Statements	H-32
5.5 Branch Statements	H-33
5.5.1 Unconditional Branches	H-33
5.5.2 Indexed Branches	H-33
5.5.3 Return Statements	H-33
5.6 Conditional Statements	H-34
5.6.1 Conditional Execution	H-34
5.6.2 Conditional Compilation	H-34
5.7 Loop Statements	H-35
Section 6 — FORMULAS	H-35
6.1 Numeric Formulas	H-36
6.2 Bit Formulas	H-37
6.3 Character Formulas	H-38
6.4 Entry Formulas	H-39

	Page
Section 7 – VARIABLES AND CONSTANTS	H-39
7.1 Variables	H-39
7.2 Constants	H-40
7.2.1 Numeric Constants	H-40
7.2.2 Bit Constants	H-41
7.2.3 Character Constants	H-41
Section 8 – FUNCTIONAL MODIFIERS	H-41
8.1 ABS functional Modifier	H-41
8.2 ENTRY Functional Modifier	H-42
8.3 NENT Functional Modifier	H-42
8.4 SHIFT Functional Modifier	H-42
8.5 INTR, BIT, BYTE Functional Modifiers	H-43
 CHAPTER II	
Section 1 – JOVIAL/J3B COMPILER UNDER OS/370	H-44
1.1 Compiling a JOVIAL/J3B Program	H-44
1.2 Assembling the Compiler Output	H-45
1.3 Link Editing and Executing the Object Program	H-45
Section 2 – SOURCE PROGRAM LISTINGS	H-46
Section 3 – ENVIRONMENT LISTING	H-46
3.1 Environment Listing Format	H-46
Section 4 – STRUCTURE OF COMPILED JOVIAL/J3B 370 PROGRAMS	H-48
4.1 The CSECT Name of the Compilation	H-48
4.2 External Conventions	H-49
4.2.1 Entry	H-49
4.2.2 Argument List	H-49
4.2.3 Return	H-49
4.3 Procedure Implementation	H-49
4.3.1 Body	H-49
4.3.2 Prologue	H-49
4.3.3 Epilogue	H-50
4.4 Basic Run-Time Support Routines	H-50
4.5 Stack Frame Management	H-50
 CHAPTER III	
Section 1 – ERROR MESSAGE CONVENTIONS AND LIST	H-52
1.1 Error Message Printout	H-52
1.2 Individual Message Format and Content	H-52
1.3 Error Message List	H-52

LIST OF FIGURES

		Page
Figure 1-1	Matrix multiple routine	H-6
Figure 1-2	Storage layout for a serial table	H-21
Figure 1-3	Storage layout for a parallel table	H-21
Figure 1-4	Storage layout for a twodimensional array	H-23
Figure 1-5	Storage layout for overlay	H-24
Figure 1-6	Source-target combinations in an <u>assignment:statement</u>	H-30
Figure 1-7	Semantic type and conversions in a numeric or relational formula involving two operands and operators	H-37
Figure 1-8	Truth tables for bit operators	H-38
Figure 2-1	Layout of a compiled program	H-48
Figure 2-2	JOVIAL/J3B 370 stack frame layout	H-51

LIST OF TABLES

		Page
Table 1-1	Semantic Types of JOVIAL/J3B	H-7
Table 1-2	Meanings of Abbreviations	H-11
Table 1-3	Meanings of Ideograms	H-12
Table 1-4	Equivalentents of Hexadecimal Digits	H-14

CHAPTER I

THE JOVIAL/J3B LANGUAGE

1. INTRODUCTION

The purpose of this chapter is to describe the language features of the JOVIAL/J3B programming language in a way that is convenient for reference. This chapter follows the same style and basic format as that used in the JOVIAL/J3B Language Specification (D229-10138). Certain of the syntax rules and explanations have been simplified as an aid to understanding. This will assist the programmer in gaining familiarity with the most common cases. It is not the intention to present a complete, formal definition of the language here. The full details of language form are contained in the specification cited above.

The information in this chapter is applicable without regard to the specific implementation of J3B or to the host computer system utilized. Implementation dependent features are discussed in the relevant target computer Language Processors Users' Guide. Operating information relevant to the host computer system is discussed in Chapter II.

The description of a JOVIAL/J3B language construct has three basic parts:

- (a) Syntax rules describing the form of the language construct. (These rules appear throughout this chapter in boxes placed at the beginning of paragraphs in which they are discussed.)
- (b) An information explanation of the meaning of the language construct.
- (c) Examples of the use of the language construct.

Figure 1-1 illustrates a complete JOVIAL/J3B program. The details of this program should be made clear by the remainder of this manual.

1.1 The Descriptive Notation

The descriptive notation used to express the syntax of JOVIAL/J3B consists of a set of rules of the form

language:construct ::= syntax:form:description.

These rules mean that the named language:construct can take the form indicated by the syntax:form:description. Throughout the manual lower case names connected by colons are used as names of the constructs in the language. In the text names of language constructs are underlined.

Syntax:form:descriptions are basically sequences of symbols from the JOVIAL language together with named language constructs. For example the rule

loop:statement ::= WHILE bit:formula ; statement

says that a loop:statement consists of the word WHILE followed by a bit:formula followed by a semicolon and a statement. In addition to the basic form a syntax:form:description can include notations for describing alternative constructions or repeated constructions. A group of alternative constructions are stacked inside curly brackets { }. One of the alternatives must be chosen. For example the rule

shift:functional:modifier ::= $\begin{Bmatrix} \text{SHIFTL} \\ \text{SHIFTR} \end{Bmatrix} (\text{bit:formula}, \text{integer:formula})$

means that a shift:functional:modifier can take on either of the two forms

SHIFTL (bit:formula , integer:formula)

or

SHIFTR (bit:formula , integer:formula)

Optional constructions are enclosed in square brackets []. One of the alternatives stacked within square brackets may be chosen, or the construction may be omitted entirely. For example the rule

single:floating:literal ::= [number] . number E_[+] number

means that in a single:floating:literal the number preceding the dot is optional and that between the E and the last number there may be a +, a - or nothing. Thus a single:floating:literal can take on one of the following forms:

```

number.numberE+number
    .numberE+number
number.numberE-number
    .numberE-number
number.numberEnumber
    .numberEnumber

```

The sign ~ following a construction indicates that constructions may be repeated one or more times. For instance the rule

```
number ::= numerical ~
```

means that a number is formed by one or more numerals. This notation may be combined with the bracket notation.

```

START "CODED BY RSE FEB. 7, 1973"

"DECLARATIONS"

ITEM II  U 31 ; "ROW COUNTER"
ITEM JJ  U 31 ; "COLUMN COUNTER"
ITEM KK  U 31 ; "INTERMEDIATE COUNTER"

CONSTANT NN  U 31 = 20 ; "ROW SIZE"
CONSTANT MM  U 31 = 10 ; "COLUMN SIZE"
CONSTANT LL  U 31 = 15 ; "INTERMEDIATE SIZE"

"PROCEDURE DEFINITION"

DEF PROC MATMUL (AA, BB : CC);
"MATRIX MULTIPLY PROCEDURE"
    BEGIN "OF MATMUL"
        "FORMAL DECLARATIONS"
        ARRAY AA (NN-1, LL-1)F;
        ARRAY BB (LL-1, MM-1)F;
        ARRAY CC (NN-1, MM-1)F;

        "TEXT"
        FOR II (0 BY 1 WHILE II <= NN-1) ;
            BEGIN "LOOP OVER ROWS OF AA"
                FOR JJ (0 BY 1 WHILE JJ <= MM-1) ;
                    BEGIN "LOOP OVER COLUMNS OF BB"
                        CC (II, JJ) = 0. ;
                        FOR KK (0 BY 1 WHILE KK <= LL-1) ;
                            "LOOP OVER COLUMNS OF AA AND ROWS OF BB"
                            CC(II, JJ) = CC(II, JJ) + AA(II, KK) * BB(KK, JJ) ;
                        END ; "OF JJ LOOP"
                    END ; "OF II LOOP"
                END ; "OF MATMUL"
            END ; "OF KK LOOP"
        END ; "OF JJ LOOP"
    END ; "OF II LOOP"
END ; "OF MATMUL"

TERM

```

Fig. 1-1 Matrix multiply routine

For instance the rule

$$\text{compound:statement} ::= \text{BEGIN} \left\{ \begin{array}{l} \text{statement} \\ \text{define:declaration} \\ \text{switch:declaration} \end{array} \right\} \sim \text{END}$$

means that a compound:statement consists of a sequence of one or more statements, define:declarations and switch:declarations mixed in any order between BEGIN and END.

The sign \sim following a construction indicates that the construction may be repeated one or more times with each occurrence separated by commas. For instance the rule

$$\text{numeric:constant:list} ::= \text{numeric:constant} \sim$$

indicates that a numeric:constant:list is a sequence of numeric:constants separated by commas.

1.2 Semantic Types

The kinds of data which can be manipulated by JOVIAL/J3B programs are organized into classes called semantic types (or data types). A given variable, name, or formula has an associated semantic type which determines what values the form may denote.

In addition to semantic types there is an alignment attribute specifying how data is stored. Semantic types and other attributes are written in all capital letters and are enclosed in angle brackets $\langle \rangle$.

There are seven basic semantic types in JOVIAL/J3B. (These are summarized in Table 1-1.) Some of these basic types are qualified with an integer which specifies a precision or length. In the following N represents an integer value.

The semantic type $\langle \text{SIGNED}(N) \rangle$ represents the set of numbers such that each number is an integer less than 2^N in magnitude, for $1 \leq N \leq 31$. A value which is $\langle \text{ALIGNED} \rangle$ occupies a full computer word; the least significant bit of the value occupies the rightmost bit of the word. A value which is $\langle \text{SIGNED}(N) \rangle$ and $\langle \text{UNALIGNED} \rangle$ occupies $N+1$ contiguous bits of a computer word, where the first (leftmost) bit represents the sign, and the remaining N bits represent the magnitude. For instance, the semantic type $\langle \text{SIGNED}(2) \rangle$ represents the numbers -3, -2, -1, 0, 1, 2, 3. A $\langle \text{SIGNED}(2) \rangle$ value which is $\langle \text{UNALIGNED} \rangle$ is stored in three contiguous bits - one for the sign and two for the magnitude. A full word used as an integer thus has semantic type $\langle \text{SIGNED}(31) \rangle$.

The semantic type $\langle \text{UNSIGNED}(N) \rangle$ represents the set of numbers such that each number is an integer greater than or equal to zero and less than 2^N , for $1 \leq N \leq 31$. A value which is $\langle \text{UNSIGNED}(N) \rangle$ and $\langle \text{ALIGNED} \rangle$ occupies a full computer word; only the rightmost N bits of the word are to be used to represent the value; the rest of the bits are assumed to be zero. A value which is $\langle \text{UNSIGNED}(N) \rangle$ and $\langle \text{UNALIGNED} \rangle$ occupies exactly N contiguous bits of a computer word; unused bits are not assumed to be any particular value. For instance the semantic type $\langle \text{UNSIGNED}(2) \rangle$ represents the numbers 0, 1, 2, 3. An $\langle \text{UNSIGNED}(2) \rangle$ value which is $\langle \text{UNALIGNED} \rangle$ is stored in two contiguous bits since there is no need for a sign bit.

TABLE 1-1

Semantic Types of JOVIAL/J3B

Semantic Types	Range of Values	Computer Implementation	
		$\langle \text{ALIGNED} \rangle$	$\langle \text{UNALIGNED} \rangle$
$\langle \text{SIGNED}(N) \rangle$	$-(2^N-1)$ to $+2^N-1$	fullword	$N+1$ bits
$\langle \text{UNSIGNED}(N) \rangle$	0 to 2^N-1	fullword	N bits
$\langle \text{SINGLE FLOAT} \rangle$	implementation defined	fullword	----
$\langle \text{DOUBLE FLOAT} \rangle$	implementation defined	doubleword	----
$\langle \text{INTEGER} \rangle$	$-(2^{31}-1)$ to $+2^{31}-1$	fullword	fullword
$\langle \text{BIT}(N) \rangle$	bit strings of length N (1 to 32)	fullword	N bits
$\langle \text{CHARACTER}(N) \rangle$	character strings of length N (0-132)	$N * 8$ bits	

The semantic type $\langle \text{SINGLE FLOAT} \rangle$ represents the set of numbers that can be represented exactly using a specified machine representation for single precision floating point numbers. $\langle \text{SINGLE FLOAT} \rangle$ values occupy a single computer word and are always $\langle \text{ALIGNED} \rangle$. Constants of semantic type $\langle \text{SINGLE FLOAT} \rangle$ (i.e., constants having

the syntactic form single:floating:constant) denote values determined by the host machine's floating point representation; otherwise, values determined by the target machine's floating point representation are implied.

The semantic type <DOUBLE FLOAT> represents the set of numbers that can be represented exactly using a specified machine representation for double precision floating point numbers.

The semantic type <INTEGER> is equivalent to <SIGNED(31)> and <ALIGNED>. Data with semantic type <INTEGER> are always full computer words.

The semantic type <BIT(N)> represents the set of strings of bits containing exactly N bits, where $1 \leq N \leq 32$. Data which is <BIT(N)> and <ALIGNED> occupies a full computer word. Data which is <BIT(N)> and <UNALIGNED> occupies N contiguous bits in a single computer word.

The semantic type <CHARACTER(N)> represents the set of strings of characters containing exactly N characters, for $0 \leq N \leq 132$. Each character is represented using an eight bit code. Characters are contiguous within computer words. The leftmost bit of a character occupies bit zero modulo 8 within a computer word, i.e., characters do not cross byte boundaries. Data of semantic type <CHARACTER(N)> is always <ALIGNED>. <CHARACTER(N)> data need not occupy an integral number of computer words. Unused bytes within a computer word are not assumed to be any particular value.

2. ELEMENTS OF THE LANGUAGE

A JOVIAL/J3B program is a structured sequence of characters which conveys a processing message to a compiler for the language. The compiler in turn produces an object program in the assembly language of a computer.

2.1 Characters and Signs

character:program ::=	character ~
character ::=	$\left\{ \begin{array}{l} \text{letter} \\ \text{numeral} \\ \text{other:marks} \end{array} \right\}$
letter ::=	$\left\{ \begin{array}{c} A \\ \vdots \\ Z \\ \$ \end{array} \right\}$
numeral ::=	$\left\{ \begin{array}{c} 0 \\ \vdots \\ 9 \end{array} \right\}$
other:marks ::=	$\left\{ \begin{array}{c} \text{space} \\ + \\ - \\ * \\ / \\ \cdot \\ = \\ , \\ (\\) \\ < \\ > \\ \vdots \\ ' \\ " \\ \% \\ \text{extra:marks} \end{array} \right\}$

2.2.1 Primitives

primitive ::=	{	ABS	}
		AND	
		ARRAY	
		BEGIN	
		BIT	
		BLOCK	
		BY	
		BYTE	
		COMPOOL	
		CONSTANT	
		DEF	
		DEFINE	
		ELSE	
		END	
		ENTRY	
		FALSE	
		FOR	
		GOTO	
		IF	
		INTR	
		INTEM	
		NENT	
		NOT	
		OR	
		OVERLAY	
		PROC	
		REF	
		RETURN	
		SHIFTL	
		SHIFTR	
		START	
		SWITCH	
		TABLE	
		TERM	
		TRUE	
		WHILE	
		XOR	

Primitives are strings of letters which are reserved symbols in the JOVIAL/J3B language. No other symbols in a program can be identical to a primitive. Each primitive has a special meaning in the language which is defined by its context in the definitions which follow.

2.2.2 Names

name ::= letter	{	letter	}	~
		numeral		
		.		

Names are programmer-assigned designations for various data objects in a JOVIAL/J3B program. Names must be distinguishable from primitives and from one another; this is achieved by unique spelling. Each unique name has one and only one meaning in a JOVIAL/J3B program. The first eight characters are used to determine uniqueness. A name must contain at least two characters. Note that a dot may be used in a name for improved readability.

2.2.3 Comments

comment ::=	{	"	}	[character]	{	"	}
		%				%	

A comment is a string of characters (excluding semicolons and quote marks) delimited at either end by a quote

mark (") or a percent sign (%). Comments have no operational effect on a program: they are used only to provide descriptive information.

Example:

"THIS IS A COMMENT"

In the examples in this manual, comments are often used to supply information about the usage of the language features illustrated by the examples. Often lower case letters are used in comments to make reading easier. Lower case letters are not, however, allowed in an actual program.

2.2.4 Abbreviations

abbreviation ::=	$\left\{ \begin{array}{c} B \\ C \\ D \\ F \\ P \\ S \\ U \end{array} \right\}$
------------------	---

Certain letters are used as abbreviations for declaring attributes of names. For example, the abbreviation B declares a name to be BIT. The meanings of these abbreviations are shown in Table 1-2.

2.2.5 Ideograms

ideogram ::=	$\left\{ \begin{array}{c} + \\ - \\ * \\ / \\ , \\ (\\) \\ < \\ > \\ : \\ : \\ ** \\ <= \\ >= \\ <> \\ = \end{array} \right\}$
--------------	--

Ideograms are the meaningful combination of special characters. They are used to represent operators in the JOVIAL/J3B language. For example, the "<>" is "not equal to", and "**" is exponentiation. The meaning of the full set of ideograms is given in Table 1-3.

TABLE 1-2

Meanings of Abbreviations

<i>Abbreviation</i>	<i>Meaning</i>
B	BIT declarer
C	CHARACTER declarer
D	DOUBLE FLOAT declarer
F	SINGLE FLOAT declarer
P	Parallel Table
S	SIGNED declarer or Serial Table
U	UNSIGNED declarer

TABLE 1-3

Meanings of Ideograms

<i>Ideogram</i>	<i>Meaning</i>
+	addition, unary plus
-	subtraction, unary minus
*	multiplication
/	division
,	list element separator
(used for grouping
)	used for grouping
<	less than
>	greater than
:	label separator, or parameter list separator
;	statement terminator
**	exponentiation
<=	less than or equal
>=	greater than or equal
<>	not equal (less than or greater than)
=	equal or assignment operator

2.2.6 Literals

$$\text{literal} ::= \begin{cases} \text{numeric:literal} \\ \text{bit:literal} \\ \text{character:literal} \end{cases}$$

JOVIAL/J3B programs contain three types of literals: numerics, strings of bits, and strings of characters. Literals denote constant values which can be determined solely by the form of the literal.

Examples

123
X'0C4'
'HI THERE'

2.2.6.1 Numeric Literals

$$\begin{aligned} \text{number} &::= \text{numeral} \sim \\ \text{numeric:literal} &::= \begin{cases} \text{single:floating:literal} \\ \text{double:floating:literal} \\ \text{integer:literal} \end{cases} \\ \text{single:floating:literal} &::= \begin{cases} \text{number}[\cdot] \\ \{ \text{number} \cdot \text{number} \} & \text{E}[\substack{+ \\ -}] \text{number} \\ \text{number} \cdot \\ \{ \text{number} \cdot \text{number} \} \end{cases} \\ \text{double:floating:literal} &::= \begin{cases} \text{number}[\cdot] \\ \{ \text{number} \cdot \text{number} \} & \text{D}[\substack{+ \\ -}] \text{number} \end{cases} \\ \text{integer:literal} &::= \text{number} \end{aligned}$$

The various formats for writing decimal numbers are denoted in the literal descriptions which follow. The permissible range of numbers depends on the semantic type (data type) of the literal.

Integer:literals, single:floating:literals and double:floating:literals denote numeric values in the conventional decimal sense. Single:floating:literals and double:floating:literals are represented by floating-point constants in the host machine

memory. The number following the E or D denotes power-of-ten exponentiation. The actual significance will be that of a single (E) or double (D) precision floating-point construct appropriate to the host machine. The range of the number is that of the host machine's floating point construct. If too much precision is specified extra digits are truncated on the right.

Integer:literals represent data of semantic type $\langle \text{UNSIGNED}(31) \rangle$. Single:floating:literals represent data of semantic type $\langle \text{SINGLE FLOAT} \rangle$. Double:floating:literals represent data of semantic type $\langle \text{DOUBLE FLOAT} \rangle$.

Examples

<i>literal</i>	<i>semantic type</i>	<i>value</i>
95.78E0	$\langle \text{SINGLE FLOAT} \rangle$	95.78
95E3	$\langle \text{SINGLE FLOAT} \rangle$	95000
6.E-2	$\langle \text{SINGLE FLOAT} \rangle$.06
.59E5	$\langle \text{SINGLE FLOAT} \rangle$	59000
4.57E+2	$\langle \text{SINGLE FLOAT} \rangle$	457
7.	$\langle \text{SINGLE FLOAT} \rangle$	7
.123	$\langle \text{SINGLE FLOAT} \rangle$.123
3.14	$\langle \text{SINGLE FLOAT} \rangle$	3.14
1.414596D0	$\langle \text{DOUBLE FLOAT} \rangle$	1.414596
75934D-2	$\langle \text{DOUBLE FLOAT} \rangle$	759.34
78.D3	$\langle \text{DOUBLE FLOAT} \rangle$	78000
.01234567D0	$\langle \text{DOUBLE FLOAT} \rangle$.01234567
49	$\langle \text{UNSIGNED}(31) \rangle$	49
769	$\langle \text{UNSIGNED}(31) \rangle$	769

2.2.6.2 Bit Literals

$\text{bit:literal} ::= \text{X'}$		numeral	\sim'
		A B C D E F	

Bit:literals denote bit patterns in a machine word. The hexadecimal number base is used for bit:literals. A full word is used to hold a bit:literal. The value is right adjusted and padded with zeroes on the left. Bit:literals represent data of semantic type $\langle \text{BIT}(32) \rangle$. The equivalence of hexadecimal digits is shown in Table 1-4.

Examples

X'3FF'
X'0C4'
X'1'

2.2.6.3 Character Literals

$\text{character:literal} ::= \text{'[character ~]'}$

Character:literals denote character strings represented by an 8-bit-per-character encoding. The maximum number of characters is 132. To represent an apostrophe in a character:literal, it is necessary to use two apostrophes to distinguish it from its use as a terminator. To help recover from errors caused by omitting the terminal apostrophe, the statement-terminator character ";" also terminates a character:literal. Thus to include a semicolon in a character literal two semicolons must be written. The absence of the terminal apostrophe is noted by a compile-time error message.

Character:literals represent data of semantic type $\langle \text{CHARACTER}(N) \rangle$ where N is the length of the character:literal.

Examples

<i>literal</i>	<i>semantic type</i>	<i>value</i>
'JOHN DOE'	$\langle \text{CHARACTER}(8) \rangle$	JOHN DOE
'X::Y'	$\langle \text{CHARACTER}(3) \rangle$	X:Y
'AB"CD'	$\langle \text{CHARACTER}(5) \rangle$	AB"CD

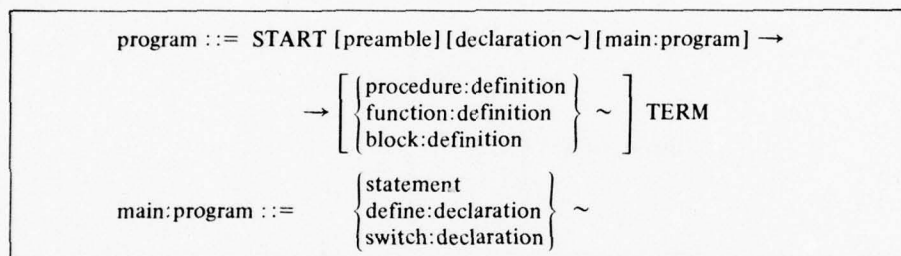
TABLE 1-4

Equivalents of Hexadecimal Digits

Hexadecimal	Decimal	Bit Pattern
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

3. FORM OF A PROGRAM

3.1 General Form



A JOVIAL/J3B program always starts with START and always ends with TERM. The preamble, if present, must immediately follow the START. It specifies the parts of the COMPOOL utilized in this program. Non-COMPOOL declarations, if any, follow the preamble. Note that in JOVIAL/J3B this is the only place declarations may appear (except for formal parameter declarations, switch declarations and define declarations). Everything declared applies to the entire program.

Presence of statements preceding procedure:definitions, function:definitions, or block:definitions indicates that the program is a main:program. The execution starts with the first statement of the main:program and ends when control is passed beyond the last statement. All parts of a program are optional except START and TERM.

There is no internal declaration scope within procedure:definitions and function:definitions so that the same name cannot be used for one purpose inside a procedure:definition and another purpose outside. However, formal:parameters, statement:names and switch:names may not be referenced outside of the procedure:definition, function:definition, or main:program in which they are defined.

Example

```

START
"preamble"
COMPOOL (TABDECL, SYMDECL);
"declaration"
ITEM II S 31;
"main program"
II = 1;
INTPROC ( );
"procedure definitions"
DEF PROC INTPROC ( );
  BEGIN
    II = II + 1;
  END;
TERM

```

3.2 Compile-Time COMPOOL

```
preamble ::= COMPOOL (file:name  $\sim$ );
file:name ::= name
```

The JOVIAL/J3B language compiler is intended to interface with a data base which is both the source of the program card images and card images comprising the compile-time COMPOOL. The compile-time COMPOOL is divided into files, each identified by file:name. Each file contains declarations which provide information commonly used by more than one program or information necessary for communication among programs. The COMPOOL is a single-source repository of such information.

By specifying a COMPOOL file name in a preamble, the programmer gets the declarations in the file processed as part of his program as though he had coded them into the program directly.

Each COMPOOL file is inserted, in sequence, immediately preceding the semicolon of the preamble. Column 1 of the first card image of the COMPOOL file is the next column to be processed after the semicolon. Each COMPOOL file is inserted in the same order in which its name appears in the preamble. Column 72 of the last card image of each COMPOOL file except the last, if any, is considered to be immediately followed by column 1 of the first card image of the next COMPOOL file. Column 72 of the last COMPOOL file is considered to be immediately followed by the column following the semicolon in the card image containing the preamble.

The syntax rules preclude nested COMPOOL files.

Any JOVIAL/J3B language form may appear in COMPOOL files, providing that the total source image, expanded as described above, is a correct JOVIAL/J3B program. COMPOOL file:names cannot be identical with primitives, but need not be distinct from other names used in program.

Example

```
START
COMPOOL (TABDECL,SYMDECL);
.
.
.
"contents of file TABDECL"
TABLE TAB (1:10) 4;
.
.
.
"contents of file SYMDECL"
ITEM SYM1 S 31;
ITEM SYM2 U 16;
.
.
.
TERM
```

3.3 Procedure and Function Definitions

```
procedure:definition ::= [DEF] PROC name (formal:parameters);  $\rightarrow$ 
 $\rightarrow$  BEGIN[formal:declarations]  $\left\{ \begin{array}{l} \text{statement} \\ \text{define:declaration} \\ \text{switch:declaration} \end{array} \right\} \sim$  END;

function:definition ::= [DEF] PROC name (formal:parameters) item:description:  $\rightarrow$ 
 $\rightarrow$  BEGIN[formal:declarations]  $\left\{ \begin{array}{l} \text{statement} \\ \text{define:declaration} \\ \text{switch:declaration} \end{array} \right\} \sim$  END;

formal:parameters ::= [name  $\sim$ ] [:name  $\sim$ ]
formal:declarations ::=  $\left\{ \begin{array}{l} \text{formal:item:declaration} \\ \text{formal:table:declaration} \\ \text{formal:array:declaration} \end{array} \right\} \sim$ 
```

The only difference between a procedure:definition and a function:definition is the appearance of an item:description of the function attributes, plus the required setting of the value of the function via an assignment to the function:name somewhere in the body of the function. There must be a formal:declaration for each formal:parameter; a formal:declaration which does not declare a formal:parameter is assumed to declare a local:parameter. Only simple items, tables and arrays are permitted as formal parameters. Function:definitions as well as procedure:definitions may have output parameters. An output parameter is one appearing to the right of the colon in the formal:parameters, while an input parameter appears to the left of the colon. If no colon is present, all of the parameters are input parameters. The option DEF is used if the procedure or function is to be invoked from outside this program.

The name appearing in a procedure:definition or function:definition takes on the syntactic form procedure:name or function:name, respectively. A function:name also takes on the more definitive syntactic form specified below, depending on the item:description.

<i>Description</i>	<i>Semantic Type</i>	<i>Syntactic Type</i>
F	<SINGLE FLOAT>	Single:floating:function:name
D	<DOUBLE FLOAT>	Double:floating:function:name
S number:bits	<SIGNED (number:bits)>	Integer:function:name
U number:bits	<UNSIGNED (number:bits)>	Integer:function:name
C number:characters	<CHARACTER (number:characters)>	Character:function:name
B number:bits	<BIT (number:bits)>	Bit:function:name

Both the procedure:definition and the function:definition serve as declarations, so that following the definition, a procedure or function may be called without using a separate declaration statement. However, since any name must be declared before it is used in a JOVIAL/J3B program, a function or procedure may be declared as well as defined in the same program to permit its use before the occurrence of the definition.

Formal:parameters, local:parameters, and statement:names defined within a function or procedure:definition may not be referenced outside of the function or procedure; however, this does not imply name scope for the parameters so defined. Each name in a JOVIAL/J3B program must be unique.

When a procedure or function is invoked execution begins with the first statement of the procedure or function definition and terminates with the execution of a RETURN statement or by passing control beyond the last statement in the definition. A procedure or function may not call itself, either directly or indirectly, i.e. JOVIAL/J3B does not allow recursion.

3.4 Block Definitions

block:definition ::=	DEF BLOCK block:name: BEGIN →
	→ [{ item:initialization table:initialization array:initialization } ~] END;
item:initialization ::=	{ integer:simple:item:name = numeric:constant; floating:simple:item:name = numeric:constant; bit:simple:item:name = bit:constant; character:simple:item:name = character:constant; }
table:initialization ::=	{ integer:table:item:name = numeric:constant:list; floating:table:item:name = numeric:constant:list; bit:table:item:name = bit:constant:list; character:simple:item:name = character:constant:list; }
array:initialization ::=	{ integer:array:name = numeric:constant:list; floating:array:name = numeric:constant:list; bit:array:name = bit:constant:list; character:array:name = character:constant:list; }
repetition ::=	integer:constant
numeric:constant:list ::=	{ numeric:constant repetition (numeric:constant) } [~]
bit:constant:list ::=	{ bit:constant repetition (bit:constant) } [~]
character:constant:list ::=	{ character:constant repetition (character:constant) } [~]

A block is a structure for grouping simple items, tables, and arrays where the intent is to allocate the physical storage for the items in a block in one program and to permit reference to them in another program. All programs that reference the block must contain a block:declaration (see 4.1.5). In addition the single program to contain the physical storage for the block must include a block:definition.

In addition to triggering the allocation of storage for the block the block:definition may optionally be used to initialize the items in the block. Only items declared in the corresponding block:declaration may be initialized.

Examples

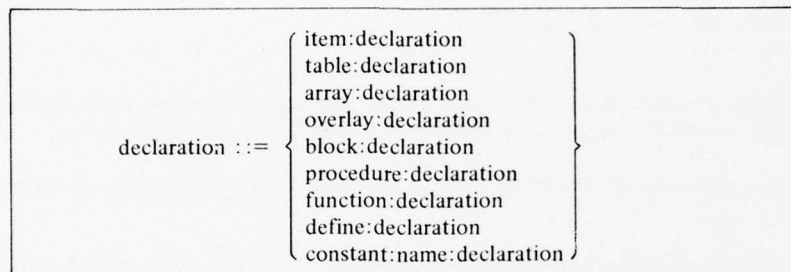
```
REF BLOCK B1; "block declaration -- appears in all programs referencing item in B1"
```

```
BEGIN
  ARRAY AA(9, 9) F;
  ARRAY BB(9, 9) F;
END
```

```
DEF BLOCK B1; "block definition -- appears in the program containing the storage for B1"
```

```
BEGIN
  BB = 100 (1.0); "initialize BB to contain all 1.0"
END
```

4. DECLARATIONS



Declarations are the means of associating programmer defined names and attributes with the elements of a program. Declarations specify the data type and form of each programmer defined name.

Every name used in a program (except COMPOOL file:names and statement:names) must be declared and except for defined:names, each name may be declared only once.

Except for formal declarations, switch declarations, and define declarations all declarations must appear at the beginning of the program before any executable statements.

Examples

```
ITEM COUNT U 31 = 0;
TABLE TAM (1.10) 2;
  BEGIN
    ITEM IV (0, 0) S 31;
  END
ARRAY AA (10, 10) D;
REF PROC FACTORIAL (U) 31;
CONSTANT ZERO S 31 = 0;
```

4.1 Data Definition and Organization

4.1.1 Item Declarations

$\text{item:declaration} ::= \text{ITEM name} \left\{ \begin{array}{l} \{F\} \quad [= \text{numeric:constant}] \\ \{D\} \\ \{S\} \quad \text{number:bits} [= \text{integer:constant}] \\ \{U\} \\ C \text{ number:character} [= \text{character:constant}] \\ B \text{ number:bits} [= \text{bit:constant}] \end{array} \right\} ;$
$\left\{ \begin{array}{l} \text{formal:item:declaration} \\ \text{local:item:declaration} \end{array} \right\} ::= \text{ITEM name item:description};$
$\text{item:description} ::= \left\{ \begin{array}{l} F \\ D \\ S \text{ number:bits} \\ U \text{ number:bits} \\ C \text{ number:characters} \\ B \text{ number:bits} \end{array} \right\}$
$\text{number:bits} ::= \text{integer:constant}$
$\text{number:characters} ::= \text{integer:constant}$

An item:declaration is used to associate attributes with an item name and to optionally assign an initial value to the item. If no initial value is explicitly assigned, the initial value is that represented by all zero bits.

Formal:item:declarations and local:item:declarations are similar to item:declarations except that no initial values may be specified.

The name following ITEM is henceforth referred to as a simple:item:name. The attributes of the simple:item:name depend on the specific form of its item:descriptions:

<i>Description</i>	<i>Semantic Type</i>	<i>Syntactic Form</i>
F	<SINGLE FLOAT>	Single:floating:simple:item:name
D	<DOUBLE FLOAT>	Double:floating:simple:item:name
S number:bits	<SIGNED (number:bits)>	Integer:simple:item:name
U number:bits	<UNSIGNED (number:bits)>	Integer:simple:item:name
C number:characters	<CHARACTER (number:characters)>	Character:simple:item:name
B number:bits	<BIT (number:bits)>	Bit:simple:item:name

The compiler may allocate a full word for an item even though its specification does not require a full word to hold the number of bits or characters specified.

Examples

```

ITEM FLOATVAL F = 1.0;
ITEM LONGVAL D = 1.414739426D0;
ITEM FIXVAL S 31;
ITEM COUNT U 31 = 0;
ITEM NAME C 20 = 'JOHN DOE';
ITEM FLAGS B4 = X'C';

DEF PROC FACTORIAL (NN);
  BEGIN
    ITEM NN U; "formal declaration"
    .
    .
  END;
```

4.1.2 Table Declarations

$$\begin{aligned}
\text{table:declaration} &::= \text{TABLE name} \left\{ \begin{array}{l} ((\text{lower:bound} :] \text{ upper:bound}) \left[\begin{array}{c} \text{S} \\ \text{P} \end{array} \right] \\ () \end{array} \right\} \rightarrow \\
&\quad \rightarrow \text{words:per:entry;} \rightarrow \\
&\quad \text{BEGIN table:item:declaration} \sim \text{END;} \\
\left\{ \begin{array}{l} \text{formal:table:declaration} \\ \text{local:table:declaration} \end{array} \right\} &::= \text{TABLE name} \left\{ \begin{array}{l} ((\text{lower:bound} :] \text{ upper:bound}) \left[\begin{array}{c} \text{S} \\ \text{P} \end{array} \right] \\ () \end{array} \right\} \\
&\quad \rightarrow \text{words:per:entry;} \rightarrow \\
&\quad \rightarrow \text{BEGIN formal:table:item:declaration} \sim \text{END;} \\
\text{table:item:declaration} &::= \text{ITEM name (starting:bit, entry:word)} \rightarrow \\
&\quad \rightarrow \text{list:item:description;} \\
\text{formal:table:item:declaration} &::= \text{ITEM name (starting:bit, entry:word)} \rightarrow \\
&\quad \rightarrow \text{item:description;} \\
\text{list:item:description} &::= \left\{ \begin{array}{l} \left\{ \begin{array}{c} \text{F} \\ \text{D} \end{array} \right\} [= \text{numeric:constant:list}] \\ \left\{ \begin{array}{c} \text{S} \\ \text{U} \end{array} \right\} \text{ number:bits [= numeric:constant:list]} \\ \text{B number:bits [= bit:constant:list]} \\ \text{C number:characters [= character:constant:list]} \end{array} \right\} \\
\text{item:descriptions} &::= \left\{ \begin{array}{c} \text{F} \\ \text{D} \\ \text{S number:bits} \\ \text{U number:bits} \\ \text{C number:characters} \\ \text{B number:bits} \end{array} \right\} \\
\text{lower:bound} &::= \text{integer:constant} \\
\text{upper:bound} &::= \text{integer:constant} \\
\text{words:per:entry} &::= \text{integer:constant} \\
\text{starting:bit} &::= \text{integer:constant} \\
\text{entry:word} &::= \text{integer:constant}
\end{aligned}$$

A table:declaration is used to specify a one-dimension structured block of storage referred to by name. A list of initial values may be specified for the items in the table. The first initial value in the list is for the first table entry, the second for the second and so on. See the syntax equation in 3.4 (block:definitions) for the form of a constant list.

Formal:table:declarations and local:table:declarations are similar to table:declarations except that no initial value list is allowed. A formal:table:declaration is used in a procedure for function definition to declare a table that is a formal parameter.

The lower:bound and upper:bound specify the indexing range to be applied to the table. The first entry in the table is numbered lower:bound and the last entry is numbered upper:bound. The number of entries is equal to the upper:bound minus the lower:bound plus one and must be one or greater. Both bounds may be negative. The default lower:bound is zero. The lower:bound must be less than the upper:bound.

If no bounds are present in the table:declaration indicated by (), the table is scalar rather than one-dimensional structured block of storage. No subscripts are permitted when referencing table:items of scalar tables, while a single subscript is required when referencing table:items of one-dimensional tables.

The words:per:entry, specifying the size of each table entry, is fixed and must be one or greater.

Table:item:declarations, specifying the attributes of the items making up each table entry, are similar to item:declarations except that a list of initial values may be specified and that data packing informations must be specified. The starting:bit and entry:word specify how the data item is positioned in the table entry. The data item is

located in the specified entry:word of the entry starting at the specified starting:bit of the word and continuing for the length of the item. The packing information may not be omitted. The bit positions in a word are numbered left to right from zero to 31. The words in an entry are numbered from zero for the first word up to words:per:entry minus one.

It is possible to specify packing information that causes items in a table entry to overlap. This can cause items to assume illegal values if an assignment is made to an overlapping component. The compiler issues a warning if it finds overlapping components. It is up to the programmer to insure that an invalid value is not accessed.

A table item of type B, S, U or F is not permitted to cross word boundaries in the table. Floating table items (F, D) must start on word boundaries (starting:bit=0). Character items (C) must start on a byte boundary (e.g., starting:bit modulo 8=0) and may (in a serial table) continue across a word boundary.

The S and P abbreviations denote serial or parallel organization respectively. The default organization is serial. Serial tables are allocated a contiguous block of storage for each table entry. Parallel tables are allocated a contiguous block for the first word of all entries, followed by a block for the second word of all entries, etc. For parallel tables, items are not permitted to cross word boundaries, due to the breaking up of entries by words. Therefore, in parallel tables, double precision items are not permitted, and character items are not permitted to extend across word boundaries.

The name following TABLE takes on the syntactic form table:name. The name following each ITEM in the table:declaration or formal:table:declaration takes on the following attributes depending on its item:description.

The alignment is <ALIGNED> for items designed F or D and <UNALIGNED> for all others except for these cases:

- (a) For S, if starting:bit equals zero and number:bits equals 31.
- (b) For B, if starting:bit equals zero and number:bits equals 32.
- (c) For C, if starting:bit modulo 8 equals zero.

The semantic type attributes are:

<i>Description</i>	<i>Semantic Type</i>	<i>Syntactic Form</i>
F	<SINGLE FLOAT>	Single:floating:table:item:name
D	<DOUBLE FLOAT>	Double:floating:table:item:name
S number:bits	<SIGNED (number:bits)>	Integer:table:item:name
U number:bits	<UNSIGNED (number:bits)>	Integer:table:item:name
C number:characters	<CHARACTER (number:characters)>	Character:table:item:name
B number:bits	<BIT (number:bits)>	Bit:table:item:name

Examples

```
TABLE      CMPLX(1:4) 2; "table of complex numbers"
BEGIN
ITEM RV (0, 0) = 1.0, 0.0, -1.0, 0.0; "real part"
ITEM IV (0, 1) F = 0.0, 1.0, 0.0, -1.0; "imaginary part"
:
:
RR = RV(2); "reference to RV component of second entry (=0.0)"
```

```
TABLE      SERTAB (2) S 5; "see Figure 1-2 for storage layout"
BEGIN
ITEM FLAGS (0, 0) B 8;
ITEM PRIOR (8, 0) S 23;
ITEM COUNT (0, 1) U 16;
ITEM POS (16, 1) S 15;
ITEM ALPHA (0, 2) C 12;
END;
```

```
TABLE      PARTAB (2) P 3; "parallel table - see Figure 1-3 for storage"
BEGIN
ITEM FLAGS (0, 0) B 8;
ITEM PRIOR (8, 0) S 23;
ITEM COUNT (0, 1) U 16;
ITEM POS (16, 1) S 15;
ITEM ALPHA (0, 2) C 4; "can't cross boundary"
```


TABLE SCALAR () 3; "scalar table"
 BEGIN
 ITEM DVAL (0, 0); "words 0 and 1"
 ITEM SVAL (0, 2) 15; "word 2 left half"
 ITEM UVAL (16, 2) U 16; "word 2 right half"
 END ;

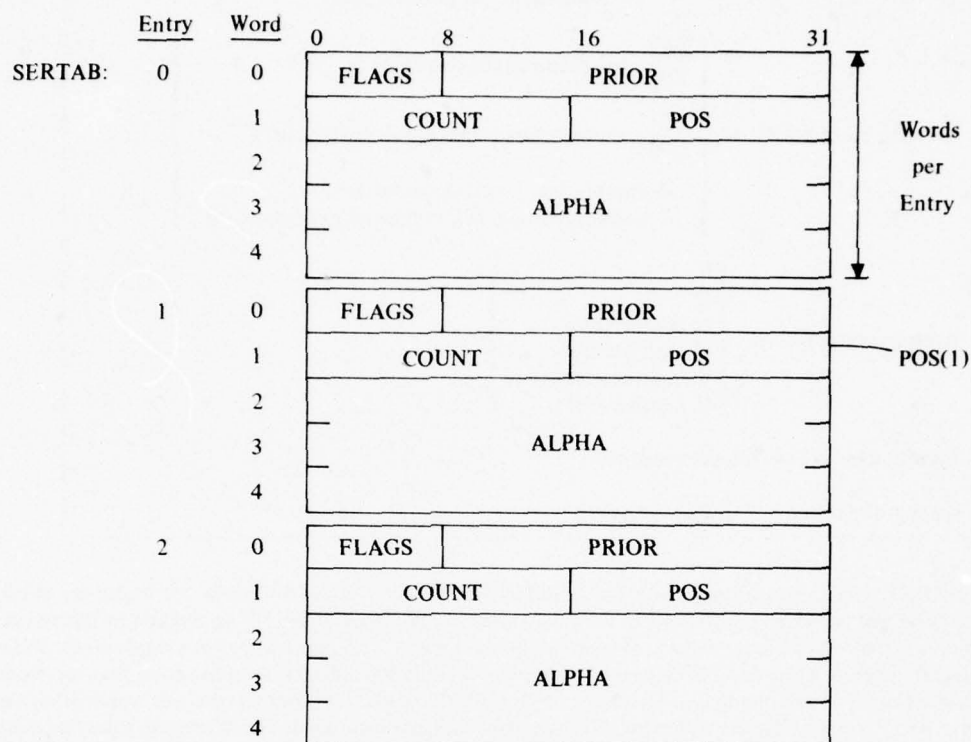


Fig.1-2 Storage layout for a serial table

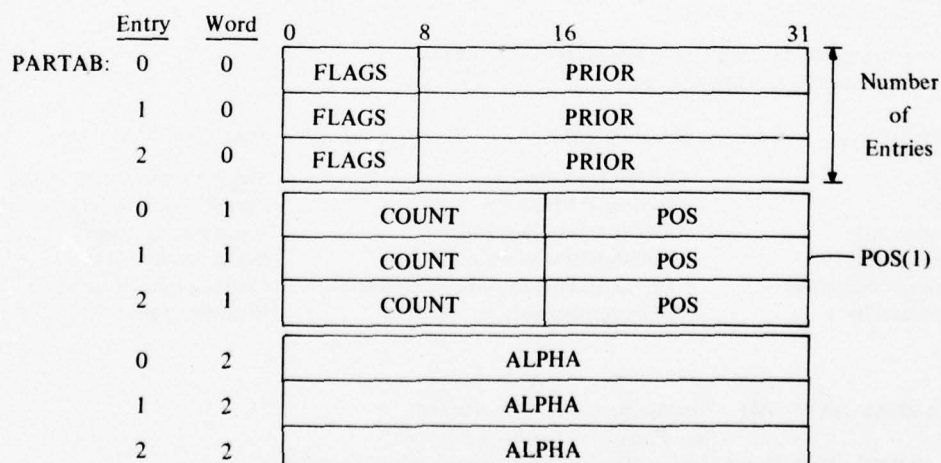


Fig.1-3 Storage layout for a parallel table

4.1.3 Array Declarations

array:declaration ::=	ARRAY name (first:dimension [, second:dimension]) → list:item:description;
formal:array:declaration ::=	ARRAY name (first:dimension[,second:→ → :dimension]) item:description;
list:item:description ::=	$\left\{ \begin{array}{l} \{F\} \\ \{D\} \end{array} \right\} \{ = \text{numeric:constant:list} \}$ $\left\{ \begin{array}{l} \{S\} \\ \{U\} \end{array} \right\} \text{number:bits} [= \text{numeric:constant:list}]$ $\left\{ \begin{array}{l} B \text{ number:bits} [= \text{bit:constant:list}] \\ C \text{ number:characters} [= \text{character:constant:list}] \end{array} \right\}$
item:description ::=	$\left\{ \begin{array}{l} F \\ D \\ S \text{ number:bits} \\ U \text{ number:bits} \\ C \text{ number:characters} \\ B \text{ number:bits} \end{array} \right\}$
first:dimension ::=	integer:constant
second:dimension ::=	integer:constant

The ARRAY declaration defines a one-or-two-dimensional array of items having the attributes specified in the item:description and optionally initialized to the values given in the constant:list. If no explicit initialization is specified, array elements are all set to binary zeroes. One and only one type of item is permitted in a single array, and each element fills an integral number of words. The organization in physical memory follows the convention that the second index varies fastest. Thus rows are contiguous in storage. For multiple word data types (D or C) all words of an element are contiguous. Array indexing begins with zero. The specified dimensions indicate the maximum subscript value. The number of array elements per column or row is the value of the dimension plus one. Each dimension must be non-negative.

The name following ARRAY takes on the syntactic form array:name. The attributes of the array:name depend on the specific item:descriptions as follows:

- (1) The alignment for each array element is <ALIGNED>.
- (2) The semantic type attributes are:

Description	Semantic Type	Syntactic Form
F	<SINGLE FLOAT>	Single:floating:array:name
D	<DOUBLE FLOAT>	Double:floating:array:name
S number:bits	<SIGNED (number:bits)>	Integer:array:name
U number:bits	<UNSIGNED (number:bits)>	Integer:array:name
C number:characters	<CHARACTER (number:characters)>	Character:array:name
B number:bits	<BIT (number:bits)>	Bit:array:name

Examples

ARRAY AA (1, 2)D; "double precision 2 by 3 array"
 "see Figure 1-4 for storage layout"
 ARRAY COUNTS (9) U 31 = 10(1); "all 10 entries initialized to one"

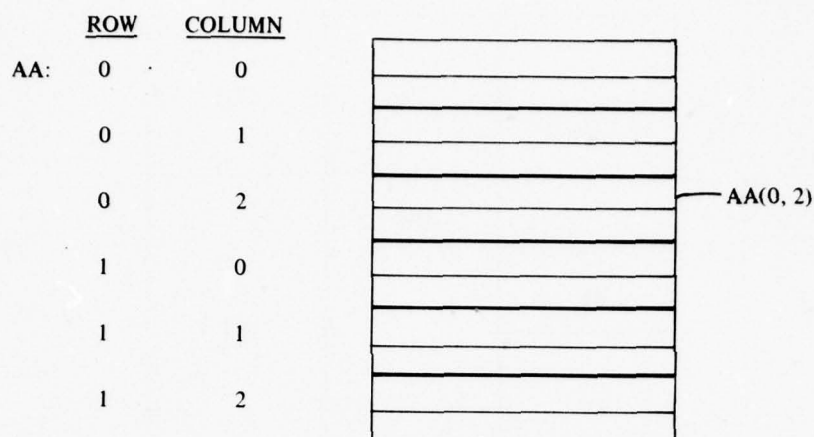


Fig.1-4 Storage layout for a two-dimensional array

4.1.4 Overlay Declarations

$$\text{overlay:declaration} ::= \text{OVERLAY element} \left\{ \begin{array}{l} \{=\} \\ \{,\} \end{array} \text{element} \right\} \sim ;$$

$$\text{element} ::= \left\{ \begin{array}{l} \text{simple:item:name} \\ \text{table:name} \\ \text{array:name} \end{array} \right\}$$

The overlay:declaration allows the programmer to specify constraints on the starting location of data elements. Two elements connected by a comma in an overlay:declaration constrain the right element to start immediately following the left element in storage. Two elements connected by an equal sign constrain the right element to start at the same storage location as the first element of the overlay:declaration.

An element may appear more than once in overlay:declarations, but any element may be constrained only once. An element in an overlay:declaration is constrained unless it is the first element of the overlay:declaration.

Elements appearing in an overlay:declaration must have been previously declared in the program. An overlay:declaration occurring within a block:declaration must contain only elements declared within that block:declaration.

Examples

OVERLAY AA, BB, CC = DD = GG, HH;

"constraints CC to follow BB, BB to follow AA,
DD to start at same location as AA,
AA is unconstrained"

"See Figure 1-5 for Storage Layout"

4.1.5 Block Declarations

$$\text{block:declaration} ::= \text{REF BLOCK name: BEGIN} \rightarrow$$

$$\rightarrow \left\{ \begin{array}{l} \text{formal:item:declaration} \\ \text{formal:table:declaration} \\ \text{formal:array:declaration} \end{array} \right\} \sim \rightarrow$$

$$\rightarrow [\text{overlay:declaration} \sim]$$

$$\text{END;}$$

A block is a structure for grouping simple items, tables and arrays where the intent is to allocate the physical storage for the items in a block in one program and to permit reference to them in another program. Blocks in JOVIAL/J3B are similar to those labeled COMMON in other programming languages. All programs that reference the block must contain a block:declaration! In addition the program containing the physical storage for the block must include a block:definition.

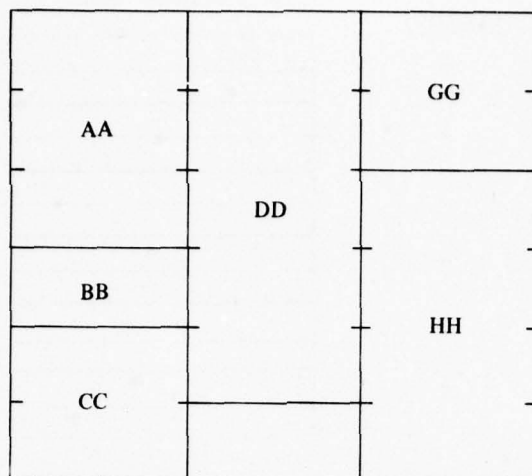


Fig.1-5 Storage layout for overlay AA, BB, CC = DD = GG, HH

The name following BLOCK takes on the syntactic form block:name. The block:name is always defined to be an external name for inter-program reference.

The data items in the block are assumed to be stored according to the order of the declarations unless overridden by overlay:declarations. The overlay:declarations may include only data items defined within the block.

When storage areas are overlayed using an OVERLAY declaration it is possible for one item to assume an illegal value caused by an assignment to an overlapping item. It is up to the programmer to insure that an invalid value is not accessed.

No initialization can be specified in a block:declaration. The block:definition (see 3.4) can be used to supply initial values for items in a block.

Examples

REF BLOCK B1; "block declaration - - appears in all programs referencing items in block B1"

```
BEGIN
  ARRAY AA(9, 9) F;
  ARRAY BB(9, 9) F;
END
```

REF BLOCK B1; "block definition - - appears in the single program containing the storage for block B1"

```
BEGIN
  BB = 100(1.0);
END ;
```


4.2 Execution Control Declarations

4.2.1 Procedure and Function Declarations

procedure:declaration ::=	REF PROC name (formal:parameter:description);
function:declaration ::=	REF PROC name (formal:parameter:description) → → item:description;
formal:parameter:description ::=	[formal:description \square] [formal:description \square]
formal:description ::=	$\left\{ \begin{array}{l} \text{formal:item:description} \\ \text{ARRAY (first:dimension [, second:dimension])} \rightarrow \\ \rightarrow \text{formal:item:description} \\ \text{TABLE } \left\{ \left(\begin{array}{l} \text{lower:bound :} \\ \text{upper:bound} \end{array} \right) \left[\begin{array}{l} \text{S} \\ \text{P} \end{array} \right] \right\} \rightarrow \\ \rightarrow \text{words:per:entry} \end{array} \right\}$
formal:item:description ::=	$\left\{ \begin{array}{l} \text{F} \\ \text{D} \\ \text{S [number:bits]} \\ \text{U [number:bits]} \\ \text{B [number:bits]} \\ \text{C [number:characters]} \end{array} \right\}$
item:description ::=	$\left\{ \begin{array}{l} \text{F} \\ \text{D} \\ \text{S number:bits} \\ \text{U number:bits} \\ \text{C number:characters} \\ \text{B number:bits} \end{array} \right\}$

The name following PROC becomes a procedure:name or function:name depending on the presence of the item:description. A function returns a value whose attributes are specified by its item:description:

- (1) The alignment is <ALIGNED>.
- (2) The semantic type attributes are:

Description	Semantic Type	Syntactic Type
F	<SINGLE FLOAT>	Single:floating:function:name
D	<DOUBLE FLOAT>	Double:floating:function:name
S number:bits	<SIGNED (number:bits)>	Integer:function:name
U number:bits	<UNSIGNED(number:bits)>	Integer:function:name
C number:characters	<CHARACTER(number:characters)>	Character:function:name
B number:bits	<BIT (number:bits)>	Bit:function:name

Procedure:declarations and function:declarations are required when the procedure is used in a program other than that in which it was defined or when the procedure is used before it is defined in a program. They are not required if the procedure:definition or function:definition appears in the same program before the first use.

The formal:parameter:descriptions provide information the compiler will use to check the semantic types of the actual:parameters in calls within the current program. Each actual:parameter used in the call of a procedure or function must match the semantic type of its corresponding formal:description. If a procedure:declaration or function:declaration is not included for a procedure:name or function:name, but a procedure:definition or function:definition is included, each actual:parameter must match the semantic type of the corresponding formal:parameter. The allowed semantic type of actual:parameters described by each formal:description is

Description	Allowed Semantic Type of <u>actual:parameter</u>
F	<SINGLE FLOAT>
D	<DOUBLE FLOAT>
S	<SIGNED (1 to 31)>
	or <UNSIGNED (1 to 31)>
S number:bits	<SIGNED (number:bits)>

Description	Allowed Semantic Type of <u>actual:parameter</u>
U	<SIGNED (1 to 31)> or <UNSIGNED (1 to 31)>
U number:bits	<UNSIGNED (number:bits)>
C	<CHARACTER (1 to 132)>
C number:characters	<CHARACTER (number:characters)>
B	<BIT (1 to 32)>
B number:bits	<BIT (number:bits)>

In addition for ARRAY, the first:dimension and second:dimension (if any) must match exactly and, for TABLE, the upper:bound, lower:bound, words:per:entry and the S/P designation must match exactly.

Examples

```
REF PROC SCAN (U:C 20,S);
ITEM NAME C 20;
ITEM TYPE S 31;
.
.
SCAN (2:NAME, TYPE);
REF PROC FACTORIAL(U) S 31;
.
.
NN = FACTORIAL (4);
REF PROC SEARCH (TABLE(1:10)4, C20);
```

4.2.2 Switch Declarations

switch:declaration ::= SWITCH name = (statement:name \sim);

The name following SWITCH takes on the syntactic form switch:name.

The switch:declaration defines an ordered list of statement:names to which execution control may pass in an indexed branch:statement. The statement:name positions are numbered successively starting with zero for the first position up to the number of statement:names minus one for the last position.

The switch:declaration is one of the few types of declarations which do not have to appear at the beginning of the program. If the switch:declaration occurs in the main:program, each of the statement:names must be defined in the main:program. If the switch:declaration appears in a procedure:definition or function:definition, each of the statement:names appearing in the switch:declaration must be defined in the procedure:definition or function:definition in which the switch:declaration occurs.

Examples

```
BEGIN
.
.
.
L0: . . . "transfers here when II=0"
.
.
.
SWITCH SW = (L0, L1, L2);
.
.
.
L2: . . . "transfers here when II=2"
L1: . . . "transfers here when II=1"
.
.
.
GOTO SW (II);
.
.
.
END ;
```

4.3 Name-Defining Declarations

4.3.1 Definition Declarations

$$\text{define:declaration} ::= \text{DEFINE name "[symbol ~]" ;}$$

Define:declarations may be used to tailor the notation used in a program in special ways. A define:declaration establishes an equivalence between a name (referred to as a defined:name) and the string of symbols between quotes. The symbol string is substituted for the defined:name if it subsequently appears as a symbol (except in a subsequent define:declaration). The following rules apply:

- (a) Comments may not appear in the definition.
- (b) Circular definitions are illegal.
- (c) If no symbols appear in the symbol string, the defined:name is effectively deleted from the source input.
- (d) The maximum number of symbols permitted is implementation dependent.
- (e) A defined:name may be redefined at a later point in the program. The textually latest definition will always apply, except when conditional compilation causes such a definition to be ignored.
- (f) A name may not be defined if it has previously been declared in the program. (A name may be defined if it has never appeared before, or if it has appeared only in define:declaration and preamble.)
- (g) Define:name substitution will not take place with a define:declaration.

A define:declaration is an exception to the rule that declarations may appear only at the beginning of a program. A define:declaration may appear intermixed with statements as well as at the beginning of a program.

Examples

```

DEFINE TABREF "TAB(II+1)";
.
.
.
TABREF = TABREF + 1; "expands to
                      TAB(II+1) = TAB(II+1)+1"
DEFINE SPELLING "SPLCMP (VOCREF (TAB(II)))";
.
.
.
PRINTIT (SPELLING); "expands to
                    PRINTIT (SPLCMP(VOCREF(TAB(II))))"
DEFINE INT "S 31";
.
.
.
ITEM COUNT INT = 1; "expand to
                    ITEM COUNT S 31 = 1"

```

4.3.2 Constant Name Declarations

$$\text{constant:name:declaration} ::= \text{CONSTANT name} \left\{ \begin{array}{l} \text{floating:description} = \text{numeric:constant} \\ \text{integer:description} = \text{integer:constant} \\ \text{B number:bits} = \text{bit:constant} \\ \text{C number:characters} = \text{character:constant} \end{array} \right\} ;$$

$$\text{floating:description} ::= \left\{ \begin{array}{l} \text{F} \\ \text{D} \end{array} \right\}$$

$$\text{integer:description} ::= \left\{ \begin{array}{l} \text{S} \\ \text{U} \end{array} \right\} \text{ number:bits}$$

The constant:name:declaration is a means of associating a name with a constant value. This improves program readability and allows modifications to be made more easily. Constant:names must be defined in a program prior to their use. They may not be redefined. Computations required to establish the value of a constant:name are performed according to normal computation rules for formulas in the host computer. Constant:names may appear in subsequent constant:name:declarations or wherever else a constant is allowed.

Examples

```

CONSTANT PI F = 3.14159;
:
:
ITEM THETA F = PI;
ITEM PHI F = 2 * PI;

CONSTANT TABSIZE U 31 = 100;
CONSTANT ENTSIZE U 31 = 4;
:
:
TABLE TAB (1: TABSIZE) ENTSIZE;
:
:

```

The name following CONSTANT takes on the syntactic form constant:name. The specific attributes depend on the form of the definition:

- (1) The alignment is <ALIGNED>.
- (2) The data type attributes are:

<i>Description</i>	<i>Semantic Type</i>	<i>Syntactic Form</i>
F	<SINGLE FLOAT>	Single:floating:constant:name
D	<DOUBLE FLOAT>	Double:floating:constant:name
S number:bits	<SIGNED(number:bits)>	Integer:constant:name
U number:bits	<UNSIGNED(number:bits)>	Integer:constant:name
C number:characters	<CHARACTER(number:characters)>	Character:constant:name
B number:bits	<BIT(number:bits)>	Bit:constant:name

5. STATEMENTS

statement ::= [name :] ~	$\left\{ \begin{array}{l} \text{null:statement} \\ \text{compound:statement} \\ \text{assignment:statement} \\ \text{call:statement} \\ \text{branch:statement} \\ \text{conditional:statement} \\ \text{loop:statement} \end{array} \right\}$
----------------------------	--

Statements are operational units, specifying data manipulation and execution sequencing. There are seven types of statements: null:statements, compound:statements, assignment:statements, call:statements, branch:statements, conditional:statements and loop:statements.

All statements are terminated with a semicolon. Statements are executed sequentially unless the sequence is modified by a call, branch, conditional or loop statement.

A statement may optionally be preceded by one or more statement names (labels) set off by colons. A statement name identifies the statement so that control may be transferred to it by a branch statement. A statement name may be used to identify only one statement in a program. (Exception: when the conditional compilation feature is used a name may label more than one statement if only one is compiled.)

Examples

```

HERE:   CNT = CNT + 1;
THERE:  YONDER: TEMP=98.6;
:
:
:
GOTO HERE;

```

5.1 Null Statement

```

null:statement ::= ;

```


The null statement, consisting of only the terminating semicolon, is a no-operation statement. It may be used, for example, as a point to set a label or to indicate that the nothing is to be done in some branch of a conditional statement.

Examples

```
DEST: ;
IF AL < 10;
    ; "do nothing"
ELSE AL = 0;
```

5.2 Compound Statements

$\text{compound:statement} ::= \text{BEGIN} \left\{ \begin{array}{l} \text{statement} \\ \text{define:declaration} \\ \text{switch:declaration} \end{array} \right\} \sim \text{END};$
--

A compound:statement consists of a group of statements surrounded by BEGIN-END. A compound:statement allows an entire group of statements to be considered as a single statement. This is especially useful in the body of conditional or loop statements. In addition to statements, define:declarations and switch:declarations may appear in a compound:statement.

Examples

```
IF VAL1 = VAL2;
    BEGIN "compound statement 1"
        VAL1 = VAL1 + VAL2;
        VAL2 = VAL1 + VAL2/2;
    END; "compound statement 2"
ELSE BEGIN "compound statement 2"
    VAL1 = VAL2
    VAL2 = 2*VAL2;
END; "compound statement 2"
```

5.3 Assignment Statement

$\text{assignment:statement} ::= \left\{ \begin{array}{ll} \text{numeric:variable} & = \text{numeric:formula;} \\ \text{bit:variable} & = \text{bit:formula;} \\ \text{character:variable} & = \text{character:formula;} \\ \text{entry:variable} & = \text{entry:formula;} \end{array} \right\}$

The assignment:statement is the basic data manipulation statement. It consists of a possibly subscripted variable, an equal sign, a formula and the terminating semicolon. The variable on the left side of the equal is referred to as the target variable while the value of the formula on the right of the equal is referred to as the source value. The assignment:statement causes the target variable to assume a value determined by the source value.

In some case, when the semantic type of the target variable differs from the semantic type of the source value, not all combinations of target variable and source value are allowed. Figure 1-6 summarizes the allowable combination and the conversion performed in each use.

In the body of a function definition, the function:name may be used as a target variable. The value assigned to the function:name when the function returns becomes the value of the function.

An input formal parameter may not be used as a target variable in an assignment statement.

There are four classes of assignment:statements divided according to the types of the target variable and source value – numeric, bit, character and entry. The special considerations applicable to each of these classes are discussed below.

5.3.1 Numeric Assignment Statements

Numeric assignment statements assign the value of a numeric formula to a numeric variable. A numeric variable designates quantities of semantic type <UNSIGNED(N)>, <SIGNED(N)>, <SINGLE FLOAT> or <DOUBLE FLOAT>. A numeric formula has one of the following semantic types – <INTEGER>, <SINGLE FLOAT> or <DOUBLE FLOAT>. The conversions applicable to numeric assignment statements are summarized in the numeric sections of Figure 1-6. Floating point values (<SINGLE FLOAT>, <DOUBLE FLOAT>) are converted to fixed point values

Type of Target Variable	Type of Source Value (SV)					
	NUMERIC			BIT(S)	CHARACTER(S)	ENTRY
	INTEGER	SINGLE FLOAT	DOUBLE FLOAT			
N UNSIGNED(T)	if $0 \leq SV \leq 2^T - 1$ then direct else undef.	truncate to INTEGER then apply INTEGER rules	truncate to INTEGER then apply INTEGER rules			
U						
M SIGNED(T)	if $(2^T - 1) \leq SV \leq (2^T - 1)$ then direct else undef.	truncate to INTEGER then use INTEGER rules	truncate to INTEGER then use INTEGER rules			
E						
R SINGLE FLOAT	exact conv. if possible otherwise approx.	direct	approx			
I						
C DOUBLE FLOAT	exact conv.	exact conv.	direct			
BIT(T)				take rightmost T bits, pad with 0 if necessary		
CHARACTER(T)					take leftmost T characters, pad with blank if necessary	
ENTRY						if word size is same then direct else undef.

Scored box:
 undef: prohibited combination
 approx: result of conversion is an approximation of the source value
 direct: no conversion is needed
 exact conv.: result is an exact equivalent of the source value

Fig. 1-6 Source-target combination in an assignment statement

($\langle \text{UNSIGNED}(N) \rangle$), $\langle \text{SIGNED}(N) \rangle$) by truncating the fractional part so that the absolute value of the target fixed point number is less than or equal to the floating point source value (truncation toward zero) e.g., 5.9 is truncated to 5 and -5.9 is truncated to -5. Fixed point numbers are converted to floating point values exactly if possible. It may not be possible to represent fixed point values (greater than some implementation defined maximum) exactly as a $\langle \text{SINGLE FLOAT} \rangle$ quantity because of the size restrictions on the mantissa. In this case the floating value will be an approximation to the fixed value.

Variables of semantic type $\langle \text{UNSIGNED}(N) \rangle$ can hold values in the range 0 to $2^N - 1$, while variables of semantic type $\langle \text{SIGNED}(N) \rangle$ can hold values in the range $-(2^N - 1)$ to $+(2^N - 1)$. The result of assigning an out-of-range value to a fixed point variable is undefined.

Examples

```
DELTA = 4*PI + ABS(EPS**2 - FUNC(ARG));
AA(II, JJ) = AA(II, JJ) + BB(II, KK) * CC(KK, JJ);
COUNT = COUNT + 1
RVAL = IVAL;
IVAL = RVAL;
```

5.3.2 Bit Assignment Statements

A bit assignment statement assigns a source value with semantic type $\text{BIT}(S)$ to a target variable of semantic type $\text{BIT}(T)$. If the bit strings are of different lengths, the rightmost T bits are assigned to the target variable. The source value is padded with zeros on the left if necessary. Note that the opposite rule is used for character string assignments.

Examples

```
FLAGS = FLAGS AND MASK;
HIGH ALT = ALT > 25000;
```

<i>target</i> <i>semantic</i> <i>type</i>	<i>source</i> <i>semantic</i> <i>type</i>	<i>source</i> <i>value</i>	<i>target</i> <i>value</i>
BIT(8)	BIT(8)	X'D9'	X'D9'
BIT(4)	BIT(8)	X'D9'	X'9'
BIT(8)	BIT(4)	X'9'	X'09'

5.3.3 Character Assignment Statements

A character assignment statement assigns a source value of semantic type $\langle \text{CHARACTER}(S) \rangle$ to a target variable of semantic type $\langle \text{CHARACTER}(T) \rangle$. If the character strings are of different lengths the leftmost T characters of the source value are assigned to the target variable. The source value is padded on the right with blanks if necessary. Note that the opposite rule is used for bit string assignments.

Examples

```
NAME = 'JOHN DOE'
DIRECTORY (NAMENT) = NAME;
```

<i>target</i> <i>semantic</i> <i>type</i>	<i>source</i> <i>semantic</i> <i>type</i>	<i>source</i> <i>value</i>	<i>target</i> <i>value</i>
CHARACTER(4)	CHARACTER(4)	'ABCD'	'ABCD'
CHARACTER(2)	CHARACTER(4)	'ABCD'	'AB'
CHARACTER(4)	CHARACTER(2)	'AB'	'AB ' (with two trailing blanks)

5.3.4 Entry Assignment Statements

An entry assignment statement is a special type of statement used to move an entire table entry in one operation (rather than moving each component in a separate assignment statement). Any entry variable is obtained with the ENTRY functional modifier. An entry formula is either a zero or an entry variable. An assignment statement of the form

ENTRY (TAB(II)) = 0;

causes the II^{th} entry of table TAB to be set to all zero bits. An assignment statement of the form

ENTRY (TAB(II)) = ENTRY (TAB2(JJ));

causes I^{th} entry of TAB1 to be set to the values of the JJ^{th} entry of TAB2. For this type of assignment to be valid the number of words in each entry of TAB1 must be the same as in each entry of TAB2. It is not required that the sub-structure of each entry be the same.

5.4 Call Statements

call:statement ::= procedure:name (actual:parameters) ;	
actual:parameter ::= [actual:input:parameter \sim] [: actual:output:parameter]	
actual:input:parameter ::=	$\left\{ \begin{array}{l} \text{numeric:formula} \\ \text{bit:formula} \\ \text{character:formula} \\ \text{table:name} \\ \text{array:name} \end{array} \right\}$
actual:output:parameter ::=	$\left\{ \begin{array}{l} \text{numeric:variable} \\ \text{bit:variable} \\ \text{character:variable} \\ \text{table:name} \\ \text{array:name} \end{array} \right\}$

A call:statement consists of a procedure:name followed by a parenthesized list of actual:parameters and the terminating semicolon. The call:statement invokes the specified procedure, passing it the evaluated actual:parameters. When the procedure returns control, control passes to the statement following the call statement.

The list of actual parameters is divided into two parts (each of which may be empty). The first part is the list of actual input parameters consisting of formulas separated by commas. An input parameter may be a numeric:formula, a bit:formula, a character:formula, a table:name or an array:name.

The second part of the actual parameter list is the list of actual:output:parameters consisting of possibly subscripted variables separated by commas. An actual output parameter may be a numeric:variable, a bit:variable, a character:variable, a table:name or an array:name. Note that an actual output parameter may not be a formula. A formal:input:parameter supplied to a procedure may not be used as an actual:output:parameter in a call:statement within the procedure:definition.

The actual output parameter list is separated from the actual input parameter list by a colon. If there is no colon all parameters are considered to be input parameters. A procedure call with no parameter is indicated by the procedure name followed by an empty pair of parentheses.

The number and semantic type of actual input and output parameters must agree with the number and semantic type of formal input and output parameters of the procedure being called, as described in the section on procedure and function declarations. Table:names used as actual:parameters must agree exactly with respect to upper and lower bounds, number of words per entry and S/P designation with its corresponding formal parameter table name. However, table structures need not agree in any other respect. Array:names used as actual:parameters must agree with their corresponding formal parameters with respect to number of dimensions, values of the dimensions, and semantic type of array element. Arrays and tables are not interchangeable. That is, an array name may not be used as an actual:parameter corresponding to a table:name formal parameter, and vice versa.

Actual:parameters are passed by reference. This means that in processing the call:statement, the value of each parameter is calculated, and the address of a location containing this value is passed to the subroutine being called. This address may be either the address of an aligned variable or the address of a temporary location generated by the J3B processor to hold the calculated value. Aligned variables (with or without subscripts) used as actual:parameters cause the address of the variable to be passed, while formulas, function calls, and unaligned variables require a temporary address to hold the value to be passed. Note that the use of a seemingly redundant operation (such as unary + or redundant parentheses) changes what may appear to be a variable into a formula, and thus causes a temporary location to be passed rather than the actual variable address. An array:name or table:name used as an actual:parameter causes an implementation-defined address to be passed, not necessarily the address of the first element of the array or table.

Examples

```

PROCAL (INPARM1, INPARM2:OUTPARM1, OUTPARM2);
OUTPRO (FORMAT, VAR1, VAR2);
INPRO (FORMAT:VAR1, VAR2);
READIT (NITERMS+ITEM, TYPE);
INTLIZ ();

```



```

MATMUL (ARAA, ARAB:ARAC);
MAKETABS (:TABL1, TABL2);

```

5.5 Branch Statements

$\text{branch:statement} ::= \begin{cases} \text{GOTO statement:name;} \\ \text{GOTO switch:name (integer:formula);} \\ \text{RETURN;} \end{cases}$

A branch:statement interrupts the ordinary listed sequence of statement execution causing control to be passed to a new location. There are three types of branch:statement – the unconditional branch, the indexed branch and the return statement.

5.5.1 Unconditional Branches

An unconditional branch statement has the form

GOTO statement:name;

This statement causes control to be passed to the statement associated with the specified statement:name. The specified statement:name must appear as the name of some statement in the same main program or procedure or function definition. Branching into or out of a procedure or function definition is not permitted. Note that GOTO is a single symbol and no space is allowed between GO and TO.

Examples

```

GOTO LABA;
:
:
LABA: COUNT = COUNT + 1 ;

```

5.5.2 Indexed Branches

An indexed branch has the form

GOTO switch:name (integer:formula) ;

where switch:name designates a switch defined in the same procedure as the branch:statement. Recall that a switch is a list of statement:names. The indexed branch:statement transfers control to the statement:name corresponding to the value of the integer:formula. The statement:names in the switch are numbered starting with zero up to the number of names minus one.

Example

```

SWITCH SW = (LABA, LABB, LABC);
:
:
LABA : . . . ; "goes here for I = 0"
:
:
LABB : . . . ; "goes here for I = 1"
:
:
GOTO SW (I) ;
:
:
LABC : . . . ; "goes here for I = 2"

```

Like the unconditional branch, the indexed branch cannot be used to branch into or out of a function or procedure definition. The action of an indexed branch is undefined if at execution the value of the integer:formula is not in the proper range.

5.5.3 Return Statements

A return statement has the form

RETURN;

The return statement, which may appear only in a procedure or function definition, terminates the execution of that

procedure or function and returns control to the point at which the procedure or function was called. A return statement is automatically generated following the last statement in a procedure or function definition so that the return branch is done after the last statement is executed. Thus it is not necessary to include a return statement in every procedure or function definition. In a function definition a value must be assigned to the function name before returning from the function definition.

5.6 Conditional Statements

```
conditional:statement ::= IF bit:formula ; statement →
                        → [ELSE statement]
```

The conditional:statement is used to select alternative active action based on the value of a bit:formula. It is used for two slightly different purposes — conditional executions and conditional compilations. The conditional compilation capability is invoked when the bit:formula has a constant value that can be determined at compile time. When the bit:formula is not constant, conditional execution is implied.

5.6.1 Conditional Execution

A conditional:statement allows selection of alternative actions depending on the value of a bit:formula. Usually the bit:formula is a bit string of length one representing TRUE or FALSE, often resulting from a comparison of other values. If the bit:formula has length greater than one then a string containing all zeros is considered to be FALSE and any other string is considered to be TRUE.

If the value of the bit:formula is TRUE then the statement immediately following the IF clause is executed. If the value of the bit:formula is FALSE then the statement following ELSE is executed if it is present. If there is no ELSE then no action is taken when bit:formula is FALSE. Following the execution of the appropriate statement, execution continues from the point immediately following the conditional:statement.

Note that the statement following the IF clause or the ELSE can be a compound:statement. This allows a single conditional:statement to select one of two complex sequences or actions.

When conditional:statements are nested, each ELSE is matched with the nearest preceding IF at the same level of BEGIN/END nesting.

Examples

```
IF ALT > 25000 ;
  BEGIN "high altitude"
  :
  :
  END; "high altitude"
ELSE BEGIN "low altitude"
  :
  :
  END; "low altitude"
IF FLAGS AND MASK1 ;
  FLAGS = FLAGS OR MASK2;
IF COND1
  IF COND2 ;
    BEGIN "COND1 and COND2"
    :
    :
    END ;
  ELSE BEGIN "COND1 and NOT COND2"
    :
    :
    END ;
VAL = VAL + 1 ;
```

5.6.2 Conditional Compilation

When the bit:formula in a conditional:statement has a constant value which can be determined at compile time the conditional compilation capability is invoked. The difference between conditional compilation and conditional

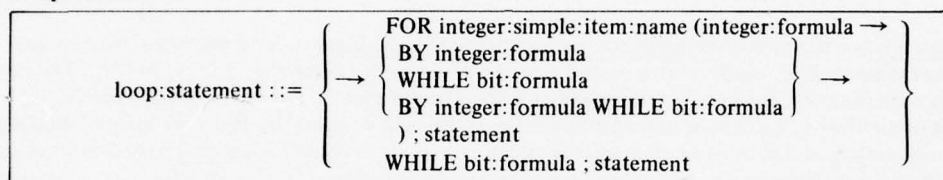
execution is that for conditional compilation the selection among alternative statements is made at compile time rather than at run time as for conditional execution. The statement that is not selected in conditional compilation is not compiled at all. The statement is checked for valid syntax but no object code is generated and no label, switch or define declarations in the statement are recognized. This means, for instance, that the same name can be used on different statements if only one of the statements is compiled.

The conditional compilations feature is especially valuable when used in conjunction with the CONSTANT declaration feature as shown in the examples.

Examples

```
CONSTANT DEBUG B1 = TRUE; "this definition may be changed to compile debug statements or not"
IF DEBUG;
    BEGIN "compiled only if DEBUG is TRUE"
LABA: INFO (COUNT) = VAL ;
    COUNT = COUNT + 2 ;
    END ;
ELSE
LABA : COUNT = COUNT + 1; "compiled only if DEBUG is FALSE"
```

5.7 Loop Statements



In both forms of loop:statement the statement which is the loop body is iterated so long as the bit:formula evaluates to other than all zero bits. The FOR form provides for establishing a variable which changes value in a prescribed orderly manner. Prior to the first iteration the value of the first integer:formula is assigned to the variable. Then, prior to each successive iteration, the current value of the second integer:formula is added to the variable. The second formula is re-evaluated with each successive iteration.

The bit:formula of the WHILE phrase is evaluated and tested before the statement is executed. Therefore, if it evaluates to FALSE at the outset, the statement in the body of the loop is never executed.

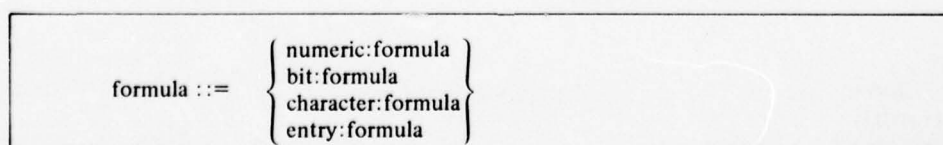
The value of the controlled variable (integer:simple:item:name) outside of the loop:statement is implementation dependent. It has no special loop:statement properties and may be used just as any other integer:simple:item:name. Transfer of control into the body of a loop:statement is undefined.

Examples

```
FOR II(1 BY 1 WHILE II <= 10) ;
    BEGIN
    TAB.CNT(II) = II ;
    TAB.FLAG(II) = X'0010' ;
    END ;

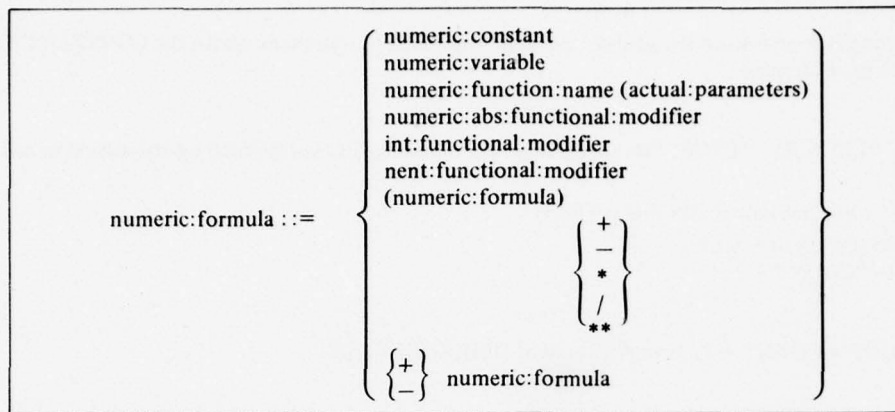
LOG2MM = 0 ;
WHILE MM > 0;
    BEGIN
    LOG2MM = LOG2MM + 1 ;
    MM = MM/2 ;
    END ;
```

6. FORMULAS



A formula is a rule for computing a value. Formulas are formed from constants, variables and function calls together with operators which specify computations on the basic values.

6.1 Numeric Formulas



A numeric:formula is composed of numeric:constants, numeric:variables and calls on functions or functional modifiers returning numeric values together with parenthesis and the numeric operator +, -, *, /, and **. The value of a numeric:formula consisting of a constant or a variable is just the value of that constant or variable. The value of a formula consisting of a call on a function or functional modifier is the value returned by that function or functional modifier. Actual:parameters passed to a function must obey the same rules as actual:parameters passed to procedures as described in section 2.5.4. Parentheses are used for grouping and do not affect the value of a formula. Computation is expressed with the numeric operators - "+" indicates additions or unary plus (which has no effect on the value); "-" indicates subtractions or negation; "*" indicates multiplication; "/" indicates division; and "**" indicates exponentiation.

The semantic type of a numeric:formula is either <DOUBLE FLOAT>, <SINGLE FLOAT>, or <INTEGER>. In a formula the signed-unsigned distinction is not maintained nor are distinctions in precision. Thus, formulas composed of semantic types <SIGNED(N)> or <UNSIGNED(N)> are considered to have semantic type <INTEGER>.

The semantic types are arranged in a hierarchy as follows:

<DOUBLE FLOAT>	highest
<SINGLE FLOAT>	↑
<INTEGER>	lowest

Any numeric:formula involving operands of different semantic types results in a numeric:formula of the type of the higher operand. The operand of the lower semantic type is converted to match the higher type. The single exception to this is that for the exponentiation operator a formula with two <INTEGER> operand results in a <SINGLE FLOAT> semantic type rather than <INTEGER>. These conversion properties are summarized in Figure 1-7.

In associating operators with operands, in the absence of parentheses the precedence of the numeric operator is as follows:

**	highest
*, /	↑
unary and binary +, -	lowest

The operator * and / have equal precedence. The operators unary +, unary -, binary +, and binary - have equal precedence. Parentheses may be used to override hierarchical order. The appearance of two or more consecutive operators is illegal. In expressions containing two or more operators of equal precedence, operators are applied in left to right order, e.g., AA+BB+CC is equivalent to ((AA+BB)+CC). Note also that AA**BB**CC is equivalent to ((A**BB)**CC).

Examples

```
COUNT + 1
AA**2 + BB**2
JJ - JJ* (II/JJ)
INTFUNC(JJ) * 5 + NENT(TABL)
-AA**2 "equivalent to -(AA**2)"
```


semantic type of right operand ROP \ semantic type of left operand LOP			
	<INTEGER>	<SINGLE FLOAT>	<DOUBLE FLOAT>
<INTEGER>	<INTEGER> (For ** <SINGLE FLOAT>)	<SINGLE FLOAT> Convert LOP	<DOUBLE FLOAT> Convert LOP
<SINGLE FLOAT>	<SINGLE FLOAT> Convert ROP	<SINGLE FLOAT>	<DOUBLE FLOAT> Convert LOP
<DOUBLE FLOAT>	<DOUBLE FLOAT> Convert ROP	<DOUBLE FLOAT> Convert ROP	<DOUBLE FLOAT>

Fig.1-7 Semantic type and conversions in a numeric or relational formula involving two operands and the operators

+, -, *, /, **,
=, <, <=, >, >=, >

6.2 Bit Formulas

bit:formula ::=	$\left\{ \begin{array}{l} \text{bit:constant} \\ \text{bit:variable} \\ \text{bit:function:name (actual:parameters)} \\ \text{shift:functional:modifier} \\ \text{NOT bit:formula} \\ (\text{bit:formula}) \\ \text{bit:formula} \left\{ \begin{array}{c} \text{AND} \\ \text{OR} \\ \text{XOR} \end{array} \right\} \text{bit:formula} \\ \text{relation:formula} \end{array} \right\}$
relation:formula ::=	$\left\{ \begin{array}{l} \text{numeric:formula} \left\{ \begin{array}{c} = \\ < \\ > \\ <> \\ <= \\ >= \end{array} \right\} \text{numeric:formula} \\ \text{character:formula} \left\{ \begin{array}{c} = \\ <> \end{array} \right\} \text{character:formula} \\ \text{entry:formula} \left\{ \begin{array}{c} = \\ <> \end{array} \right\} \text{entry:formula} \end{array} \right\}$

All bit:formulas are <ALIGNED>. The semantic type of a bit:formula is <BIT(N)> where the length N is the length of the longest bit string used as an operand. The length for a relation:formula is one. In a bit:formula involving operands of different lengths the shorter operand is padded on the left with zeros until both operands are the same length.

The bit operators NOT, AND, OR, and XOR work on each bit of a bit string separately. Figure 1-8 contains truth tables describing the value of ith bit of the result given the ith bits of the operands.

In associating operators with operands, in the absence of parentheses, the precedence of the bit operators is as follows:

NOT	highest
AND	↑
OR, XOR	lowest

where OR and XOR have equal precedence. Parentheses may be used to override this hierarchy. Where two or more operators of the same precedence occur in the same formula, they are applied in left to right order.

A relation:formula indicates that two values are to be compared resulting in a bit string of length one which is interpreted as being TRUE (X'1') if the specified relation holds and FALSE (X'0') if the specified relation does not hold. Comparison of numeric values can involve one of the following six operators:

<i>Sign</i>	<i>Relation Specified</i>
=	equals
<>	not equals (less than or greater than)
<	less than
<=	less than or equals
>	greater than
>=	greater than or equals

<i>i</i> th operand bit		<i>i</i> th result bit operator			<i>i</i> th operand bit	<i>i</i> th result bit NOT
left	right	AND	OR	XOR		
0	0	0	0	0	0	1
0	1	0	1	1	1	0
1	0	0	1	1		
1	1	1	1	0		

Fig.1-8 Truth tables for bit operators

If the numeric:formulas being compared do not have the same semantic type, the types are converted according to the hierarchy described in the previous section on numeric:formulas (see Figure 1-7). The formula with the lower semantic type is converted to the semantic type of the other formula.

Character:formulas can only be compared for equality (=) or inequality (<>). If the character strings being compared are of different lengths the shorter is padded on the right with blanks.

Entry formulas can only be compared for equality (=) or inequality (<>). Entry formulas being compared must have the same number of words.

Examples

```
NOT FLAGS AND (MASK1 OR MASK2)
REG1 XOR REG2
AA = BB AND CC <> DD OR AA <= DD
NAME1 = 'JOHN DOE'
ENTRY (TAB0 (II)) <> ENTRY (TAB(JJ))
```

6.3 Character Formulas

$\text{character:formula} ::= \left\{ \begin{array}{l} \text{character:constant} \\ \text{character:variable} \\ \text{character:function:name(actual:parameters)} \\ \text{byte:functional:modifier} \end{array} \right\}$

There are no operators for manipulating character strings so a character:formula consists of a single character constant, variable, function call, or call on the BYTE functional modifier. The semantic type of a character formula is CHARACTER(N) where the length N is the length of the constant, variable or function value composing the formula. The length of a character:formula consisting of a call on the BYTE functional modifier is the number:characters specified in the call. All character:formulas are <ALIGNED>.

Examples

```
CHARTAB (JJ)
'JOHN DOE'
LEXFUNC (ARG1:ARG2)
BYTE (STRING, 3, 2)
```

6.4 Entry Formulas

$$\text{entry:formula} ::= \begin{cases} 0 \\ \text{entry:variable} \end{cases}$$

An entry:formula identifies an entire table entry (entry:variable) or a dummy table entry whose bits are all zero (0). Entry:formulas can only be assigned to entry:variables or compared with other entry:formulas. The length of an entry:variable (which is specified with the ENTRY functional modifier) is the number of words in the entry. The entry formula 0 is considered to be as long as necessary to match the length of the entry variable it is being assigned to or compared with.

Examples

```
ENTRY (TAB(II))
ENTRY (TAB (JJ+1))
0
```

7. VARIABLES AND CONSTANTS**7.1 Variable**

$$\begin{aligned} \text{numeric:variable} &::= \begin{cases} \text{double:floating:variable} \\ \text{single:floating:variable} \\ \text{integer:variable} \end{cases} \\ \text{double:floating:variable} &::= \begin{cases} \text{double:floating:simple:item:name} \\ \text{double:floating:table:item:name [(subscript)]} \\ \text{double:floating:array:name (subscript [, subscript])} \end{cases} \\ \text{single:floating:variable} &::= \begin{cases} \text{single:floating:simple:item:name} \\ \text{single:floating:table:item:name [(subscript)]} \\ \text{single:floating:array:name (subscript [, subscript])} \end{cases} \\ \text{integer:variable} &::= \begin{cases} \text{integer:simple:item:name} \\ \text{integer:table:item:name [(subscript)]} \\ \text{integer:array:name (subscript [, subscript])} \end{cases} \\ \text{bit:variable} &::= \begin{cases} \text{bit:simple:item:name} \\ \text{bit:table:item:name [(subscript)]} \\ \text{bit:array:name (subscript [, subscript])} \end{cases} \\ \text{character:variable} &::= \begin{cases} \text{character:simple:item:name} \\ \text{character:table:item:name [(subscript)]} \\ \text{character:array:name (subscript [, subscript])} \end{cases} \\ \text{subscript} &::= \text{integer:formula} \\ \text{entry:variable} &::= \text{entry:functional:modifier} \end{aligned}$$

A variable is a named location suitable for storing a value. A variable is either a simple:item:name, a possible subscripted table:item:name or a subscripted array:name. The semantic type and alignment attributes of a variable are the semantic type and alignment implied by the declaration of its name.

Table:item:names must be subscripted unless the table was declared to be a scalar. Array:names must have the same number of subscripts as specified in the declaration. Results are undefined if any subscript is not in the declared range.

Before execution the value assumed by a variable is the declared initial value or if there is no initial value then it is the value represented by all zero bits. The current value of a variable is the last value assigned to it with an assignment statement.

Entry variables can only be formed by using the ENTRY functional modifier.

Examples

```
AA
TAB(II + 1)
ARA (II, JJ + KK)
ENTRY (TAB(II + 1))
```

7.2 Constants

$\text{constant} ::= \left\{ \begin{array}{l} \text{numeric:constant} \\ \text{bit:constant} \\ \text{character:constant} \end{array} \right\}$

Constants denote nonvarying data values.

7.2.1 Numeric Constants

$\text{numeric:constant} ::= \left\{ \begin{array}{l} \text{numeric:constant:name} \\ \text{numeric:literal} \\ \text{numeric:constant} \left\{ \begin{array}{l} + \\ - \\ * \\ / \end{array} \right\} \text{numeric:constant} \\ \left\{ \begin{array}{l} + \\ - \end{array} \right\} \text{numeric:constant} \\ (\text{numeric:constant}) \end{array} \right\}$

A numeric:constant is basically a numeric:literal (see 2.2.6.1) or a name declared to be a numeric:constant using the CONSTANT declaration.

In addition any numeric:formula having a constant value (except those containing the operator **) may be used anywhere a constant is required. The value of a constant numeric:formula is computed by the rules of numeric:formula at compile time using the arithmetic of the host computer.

Examples

```
1
1.414
1.414293767D0
6.23E10
7.414294314E-7
"assume the following declaration"
CONSTANT PI F = 3.14159 ;
PI
2 * PI
PI/2
```


7.2.2 Bit Constants

$\text{bit:constant} ::=$	$\left\{ \begin{array}{l} \text{bit:constant:name} \\ \text{TRUE} \\ \text{FALSE} \\ \text{bit:literal} \\ \text{NOT bit:constant} \\ (\text{bit:constant}) \\ \text{bit:constant} \left\{ \begin{array}{l} \text{AND} \\ \text{OR} \\ \text{XOR} \end{array} \right\} \text{bit:constant} \end{array} \right\}$
---------------------------	--

A bit:constant is a bit:literal (see 2.2.6.2), a name declared to be a bit:constant using the **CONSTANT** declaration, the special bit:constants **TRUE** and **FALSE** or a bit:formula having a constant value. The bit constants **TRUE** and **FALSE** behave as though the following declarations had appeared in the program:

```
CONSTANT TRUE B 1 = X'1';
CONSTANT FALSE B 1 = X'0';
```

The value of a constant bit:formula is computed at compile time by the rules for the evaluation of bit:formulas. Constant bit:formulas are especially useful in conjunction with the conditional compilation facility.

Examples

```
"assume the following declarations"
CONSTANT OPTION1 B 1 = TRUE ;
CONSTANT OPTION2 B 1 = FALSE ;

OPTION 1 AND NOT OPTION 2
TRUE
X'FC' AND TRUE
X'12AF'
```

7.2.3 Character Constants

$\text{character:constant} ::=$	$\left\{ \begin{array}{l} \text{character:constant:name} \\ \text{character:literal} \end{array} \right\}$
---------------------------------	--

A character:constant is either a character:literal (see section 2.2.6.3) or a name declared to be a character constant using the **CONSTANT** declaration.

Examples

```
CONSTANT PROGNAME C 20 = 'LEXFUNC' ;
PROGNAME
'JOHN DOE'
```

8. FUNCTIONAL MODIFIERS

Functional modifiers are built-in functions used to obtain special kinds of information about data items and structures and to do special types of processing. Functional modifiers resemble function calls in form and, like function calls, they return values. Some functional modifiers (e.g., **ENTRY**) may be treated like variables and can appear on the left side of an assignment statement.

8.1 ABS Functional Modifier

$\text{numeric:abs:functional:modifier} ::=$	ABS (numeric:formula)
--	------------------------------

The abs:functional:modifier computes the absolute value of the numeric formula given as argument. The semantic type of the **ABS** functional modifier is the same semantic type as the formula within the parentheses.

Examples

```
ABS (FTVAL)
ABS (BB**2 - 4*AA*CC)
```

8.2 ENTRY Functional Modifier

$$\text{entry:functional:modifier} ::= \text{ENTRY (table:name(subscript))}$$

The ENTRY modifier allows a table entry to be treated as a single unit. The subscript distinguishes the entry from others in the table.

The entry:functional:modifier has for its value a particular table entry designated by the subscript value. The subscript is omitted for scalar tables. The value is undefined if the subscript value is less than the lower:bound of the table or greater than the upper:bound.

The ENTRY functional modifier may be used on the left hand side of an entry assignment statement.

Examples

```
ENTRY (TAB(II+1))
ENTRY (SCALTAB)
```

8.3 NENT Functional Modifier

$$\text{nent:functional:modifier} ::= \text{NENT (table:name)}$$

The value of the nent:functional:modifier is the number of entries in the specified table. The nent:functional:modifier has the semantic type <INTEGER>.

Examples

```
NENT (TAB)
NENT (SCALTAB)
```

8.4 SHIFT Functional Modifier

$$\text{shift:functional:modifier} ::= \left\{ \begin{array}{l} \text{SHIFTL} \\ \text{SHIFTR} \end{array} \right\} (\text{bit:formula}, \text{integer:formula})$$

The shift:functional:modifier causes the bit:formula to be shifted the number of positions specified by the integer:formula. A left shift is indicated by SHIFTL while a right shift is indicated by SHIFTR.

The shift:functional:modifier has semantic type <BIT(32)> and is <ALIGNED>.

For a left shift, zeros are shifted in on the right and any bits shifted left beyond the beginning of the word (bit position 0) are lost. For a right shift zeros are shifted in on the left and any bits shifted right beyond the end of the word (bit position 31) are lost. A shift of more than 32 results in a string of all zero bits. The result is undefined if the integer:formula indicating the number of positions to shift is negative.

Examples

```
SHIFTL (X'000F', 4) "=X'00F0' "
SHIFTL (X'12345678', 8) "=X'34567800' "
SHIFTR (X'12345678', 8) "=X'00123456' "
SHIFTR (X'CCF0', 40) "=X'0' "
```

8.5 INTR, BIT, BYTE Functional Modifiers

```

int:functional:modifier ::= INTR (named:variable, index, number:bits)

bit:functional:modifier ::= BIT(named:variable, index, number:bits)

byte:functional:modifier ::= BYTE(named:variable, index, number:characters)

index ::= integer:formula

named:variable ::= {
    simple:item:name
    table:item:name [(subscript)]
    array:name (subscript [, subscript])
}

```

This group of functional modifiers allows data items to be decomposed into bits or bytes. They also allow data items of one semantic type to be treated as data items of another semantic type.

For the INTR and BIT functional modifiers the named variable is considered to be a string of bits whose length depends on the semantic type of the variable as follows:

<i>semantic type</i>	<i>assumed bit string length (L)</i>
<UNSIGNED(N)>	N
<SIGNED(N)>	N+1
<SINGLE FLOAT>	32
<DOUBLE FLOAT>	64
<BIT(N)>	N
<CHARACTER(N)>	8*N

The INTR and BIT functional modifiers select a substring starting with the bit position indicated by the second parameter index and continuing for the specified number:bits. The bits are numbered starting from zero at the left end of testing for named:variable. Results are undefined if the specified substring is not entirely contained in the named:variable. For the INTR functional modifier the selected bits are considered to be an integer of semantic type <UNSIGNED(S)> where S is the value of number:bits (which must be non-negative and less than or equal to 31). For the BIT functional modifier the selected bits are considered to form a bit string of semantic type <BIT(S)> where S is the value of number:bits (which must be non-negative and less than or equal to 32). Note that the selected string may cross a word boundary as long as the total length is less than the maximum (31 for INTR and 32 for BIT).

For the BYTE functional modifier the named:variable is considered to be a character string whose length is the number of 8 bit characters that fit entirely within the named:variable. This length depends on the semantic type as follows (all fractional parts in the division indicated below are truncated):

<i>semantic type</i>	<i>assumed character string length (L)</i>
<UNSIGNED(N)>	N/8
<SIGNED(N)>	(N+1)/8
<SINGLE FLOAT>	4
<DOUBLE FLOAT>	8
<BIT(N)>	N/8
<CHARACTER(N)>	N

The BYTE functional modifier selects a substring starting with the character position indicated by the second parameter index and continuing for the specified number:characters. Results are undefined if the specified substring is not entirely contained in named:variable. The semantic type of the BYTE functional modifier is <CHARACTER(S)> where S is the value of number:characters (which must be non-negative and less than or equal to 132). Note that the BYTE functional modifier does not allow access to an incomplete high order byte.

Examples

```

INTR (BITVAR, 0, 31)
INTR (CHSTG, 16, 8)
BIT (INTVAR, 0, 32)
BIT (BITVAR, 7, 1)
BIT (DVAR, 30, 30)
BYTE (BITVAR, 8, 3)
BYTE (CHSTG, 6, 1)

```

CHAPTER II

USE OF THE JOVIAL/J3B COMPILERS UNDER OS/370

The JOVIAL/J3B Compilers translate JOVIAL/J3B source programs into assembly language code for a target computer. The JOVIAL/J3B 370 Compiler produces BAL assembly language code for the IBM System 370. Other implementations produce assembly language for specific avionics computers. The following discussion is limited to the JOVIAL/J3B 370 Compiler.

The following sections describe the B-1 Support Software and OS/370 job control information required to use the JOVIAL/J3B 370 compiler. Full details on use of the B-1 Support Software are contained in B-1 Avionics Support Software Control Program/Software Management Processors Users: Guide (D229-10049-1).

1. JOVIAL/J3B COMPILER UNDER OS/370

The JOVIAL/J3B 370 compiler operates on an IBM 370 computer under control of the B-1 Support Software (SS). A 280 Kbyte region of core storage is sufficient to compile moderate-size programs.

1.1 Compiling a JOVIAL/J3B Program

The JOVIAL/J3B 370 Compiler is invoked by the SS Control Program upon encountering a SS control card with the processor mnemonic JOV. The card images which are to form the JOVIAL/J3B source program may immediately follow the SS control card in the system card:image:input file, may be obtained from a processor card:image:input file, or may be obtained from a SS data base. The card images comprising COMPOOL files are obtained from a SS data base. The card images comprising the object assembly language program may be output to a SS data base or may be output to a processor card:image:output file.

Options available to the JOVIAL/J3B 370 compiler are as follows:

- C List all COMPOOL files as processed.
- D Remove alignment restriction on double precision variables (370 only)
- E Print an expanded error message listing
- G Do not generate an output program
- N Do not print an environment listing
- O Print the generated object code
- T Do not print a summary statistics listing
- U Do not print the J3B source program
- X Print a set/used listing.

The JOVIAL/J3B 370 Compiler control card has the form

$$\begin{aligned} &\& \text{PRC}=\text{JOV} \left[\begin{array}{l} \text{,card:image:input} \\ \text{,input:module} \end{array} \right] \left\{ \begin{array}{l} \text{,card:image:output} \\ \text{,output:module} \end{array} \right\} \rightarrow \\ &\rightarrow [\text{,system:library}] [\text{,options}] \end{aligned}$$

card:image:input identifies the file from which the source card images may be obtained. input:module identifies the symbolic module in an SS data base which may contain the source program. If neither is specified, the source program must follow the control card.

card:image:output identifies the file into which the resultant assembly language card images may be placed. output:module identifies a symbolic module to be created on an SS data base. One of these must be specified.

system:library identifies the SS data base to be searched for COMPOOL files. If not specified, the system data base is assumed.

The form of the JOVIAL/J3B 370 Compiler control card given above is typical, but not all variations are included. Full details of processor control cards are included in D229-10049-1.

EXAMPLE 1:

& PRC=JOV, CI=CARDS,CO=J3BOBJ

In this example, the source program is read from the file indicated by CARS and the object program is written in the file designated by J3BOBJ.

& PRC=JOV,OM=J3BOBJ,SL=COMPOOL

In this example, the source program immediately follows the SS control card and the object program is placed in the system data base under the name J3BOBJ. Requested COMPOOL files are obtained from the SS data base identified by COMPOOL.

Execution of compiled programs requires an assembly and linkedit operation following compilation. The most convenient way of communicating the JOVIAL/J3B compiler output to the standard OS/370 processors is through use of the card:image:output parameter on the compilation control card. Alternatively, the LOAD directive of the Data Base Utility Processor (DUP) may be used.

If the card:image:output file is designated by J3BOBJ, the file information may be specified in the JCL as follows:

```
//J3BOBJ① DD DSN=&&J3BOBJ,DISP=(NEW,PASS),UNIT=SYSDA,
//          DCB=(RECFM=FB,BLKSIZE=1600,LRECL=80),
//          SPACE=(3200,(100,20))
```

- ① The assembly language output of the compiler is placed in the J3BOBJ file to be passed to the assembly step. Normally, J3BOBJ contains the object output from a single-compilation since in general only one program is accepted by the IBM assemblers.

1.2 Assembling the Compiler Output

The compiler object is assembled using one of the standard OS 370 assemblers. The following is an example of the Job Control for the F-level assembler, using the standard IBM ASMFC catalogued procedure.

```
// EXEC ASMFC,APARM='XREF,DECK,LOAD,LINECNT=58'①
//SYSIN②DD DSN=&&J3BOBJ,DISP=(OLD,DELETE),
//          DCB=(RECFM=FB,BLKSIZE=1600,LRECL=80)
/*
```

- ① Refer to installation information on parameters and defaults for the catalogued procedure, ASMFC.
 ② The assembler input is designated by the DDNAME SYSIN. &&J3BOBJ is the data set (DDNAME J3BOBJ) passed as the output of the compiler step.

1.3 Link Editing and Executing the Object Program

The assembly output is a standard-format TEXT (object) deck which may be link-edited with other TEXT decks using the IEWL link-editor program. To link-edit the required JOVIAL/J3B run-time support, the user must specify the AED library and other needed user libraries on concatenated DD-cards with a DDNAME of SYSLIB in the IEWL step. A sample link-edit step follows:

```
//LKED EXEC PGM=IEWL,PARM=(MAP,LET,LIST), REGION=96K
//SYSPRINT DD SYSOUT=A
//SYSLMOD DD DSN=&LMOD(USRPR), UNIT=SYSDA,DISP=(,PASS),
//          SPACE=(3072,(100,50,1),RLSE)
//SYSUTI DD UNIT=SYSDA,SPACE=(3072,(20,10))
//SYSLIB DD DSN=AED①,LIBRARY AED,DISP=SHR,UNIT=2314,VOL=SER=②
//SYSLIN DD DSN=&LOADST③, UNIT=SYSDA,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN④
//SYSIN DD *
(OBJECT DECKS AND LINKAGE EDITOR CONTROL STATEMENTS GO HERE)
/*
```

- ① Libraries on DD cards following SYSLIB will be searched in the order listed to find subroutines and resolve external references. When the Trace or Zones Free Storage libraries are used, they must precede the description of the standard AED library. The AED release number is needed here in the data set name of the AED library. A system library could be listed second.
 ② The volume serial number of the user's disk holding the AED libraries.
 ③ ④ The input to the linkage editor, SYSLIN, is described here to consist of the output of the assembler (&LOADST in this sample) followed by TEXT decks to be placed after the //SYSIN DD * card.

Execution would require a job step with an EXEC card such as

```
//GO EXEC PGRM=*.LKED.SYSLMOD,REGION=_____K.
```

All descriptions of compilation, link-edit, etc. have assumed that any given set of executable programs includes a compilation containing a main program.

A main program is compiled as an external procedure with no arguments. The external name assigned by the compiler to this external procedure is #MAIN. Execution must start at entry #MAIN.

2. SOURCE PROGRAM LISTING

The source program listing contains all program input lines processed by the compiler plus error messages for program errors detected prior to or during the production of the source program listing. It includes program input lines obtained from COMPOOL files.

The sequence number assigned by the compiler appears in the source program listing at the beginning of each input line. Any characters inserted by the compiler (e.g., expansion of DEFINED names) do not appear in the source program listing.

Error messages for program errors detected prior to or during production of the source program listing will be printed on the first available line following the source line containing the error. If more than one error is detected during the processing of an input line, each error message will be printed, no more than one message per line.

The compiler will output as a separately grouped collection, an English narrative explanation for each type of error detected in the compiled program.

The source listing contains a summary of static statistics for the program. The following static statistics are generated:

- number of characters
- number of cards
- number of items
- number of declarations
- number of comments
- number of assignment statements
- number of formulas
- formula complexity
- number of IF statements
- number of simple GOTO statements
- number of switch GOTO statements
- number of loop statements

3. ENVIRONMENT LISTING

The Environment listing is an aid to the programmer concerned with documentation, debugging, or analysis of a JOVIAL/J3B program. It provides a display of the attributes of every symbol declared in a JOVIAL/J3B program.

3.1 Environment Listing Format

The environment listing displays symbols in alphanumeric order. There is one line per symbol. Each line of output has ten columns, as explained below. However, some columns may be blank for a given symbol depending on its attributes.

Column 1 – NAME

Column 1 contains the symbol name

Column 2 – COMPOOL

Column 2 contains the name of the COMPOOL file within which the symbol is defined. This column is blank if the symbol is not defined in a COMPOOL file.

Column 3 – LINE

Column 3 contains the number of the line in which the symbol is defined. The line number is always relative within the file given in column 2.

The information printed in columns 1, 2 and 3 is standard for all symbols. The information in the remaining columns (4 through 10) varies depending on the attributes of the symbol.

Column 4 – DECLARED TYPE

Column 4 contains the declared type of the symbol. One of the following literals will appear in this column.

CONSTANT
ITEM
TABLE ITEM
ARRAY
FUNCTION
TABLE
BLOCK
PROCEDURE
STATEMENT NAME
SWITCH
DEFINED

Column 5 – DATA TYPE

Column 5 contains the data type for symbols declared as CONSTANT, ITEM, TABLE ITEM, ARRAY, or FUNCTION and is blank otherwise. When the column is not blank one of the following literals will appear.

F
D
S
U
B
C

Column 6 – SIZE

Column 6 contains the size of the symbol as follows:

if column 5 is:	column 6 is:
F	24
D	56
S	number:bits
U	number:bits
B	number:bits
C	number:characters

If column 5 is blank and column 4 is TABLE, column 6 is the number

(lower bound – upper bound + 1) * words:per:entry

If column 5 is blank and column 4 is BLOCK, column 6 is the size of block in words.

Otherwise, column 6 is blank.

Column 7 – SCOPE

Column 7 contains the scope of the symbol. One of the following literals appear in this column:

INT
DEF
REF
PAR

PAR will appear when the symbol is a formal parameter.

DEF will appear when the symbol is an external procedure, function, or block which is defined in the program.

REF will appear when the symbol is a procedure, function, or block which is just referenced in the program.

INT will appear in all other cases.

Column 8 – EXTENT

Column 8 contains the extent within which the scope of column 7 applies. One of the following strings will appear in this column.

#DATA
#MAIN
a block:name
a table:name
a procedure:name
a function:name

The string #DATA will appear for all symbols declared as CONSTANT, ITEM, ARRAY outside of a table or block.

The string #MAIN will appear for all symbols declared as DEFINED, PROCEDURE, FUNCTION, STATEMENT NAME, or SWITCH outside of any procedure or function definition.

A block:name will appear for all symbols declared within that block.

A table:name will appear for all TABLE ITEMS declared within that table.

A procedure:name or function:name will appear for all formal parameters, statement names, and switches declared within that procedure.

Column 9 – OFFSET IN EXTENT

Column 9 contains the offset of the symbol within the extent given in column 8 for symbols of declared type ITEM, ARRAY, TABLE, and BLOCK; and of scope INT, DEF, or REF. It contains the starting:bit and word:number for symbols of declared type TABLE ITEM and of scope INT; it contains the parameter number for symbols of declared type ITEM, ARRAY, and TABLE and of scope PAR; and is blank otherwise.

Column 10 – DIMENSION

Column 10 contains the dimension of bound information of the symbol. This column contains 1 for symbols of declared type ITEM, and TABLE ITEM. It contains the values of the dimension(s) for symbols of declared type ARRAY. It contains the bounds for symbols of declared type TABLE; it contains the number of switch elements for symbols of declared type SWITCH; it contains the number of parameters for symbols of declared type PROCEDURE or FUNCTION; it contains the actual value for symbols of declared type CONSTANT; and it is blank otherwise.

4. STRUCTURE OF COMPILED JOVIAL/J3B 370 PROGRAMS

A compiled JOVIAL/J3B 370 program (Fig.2-1) results in a control section (CSECT) with one or more entry points. Each entry point corresponds to a procedure defined with the JOVIAL/J3B program and available for external reference. In addition to these externally available procedures there may be any number of internal procedures that can be referenced explicitly only from within the control section.

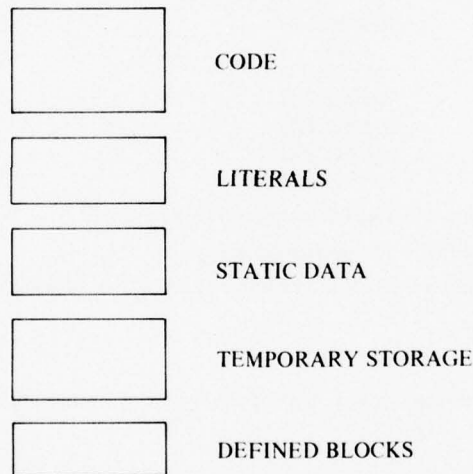


Fig.2-1 Layout of a compiled program

The first portion of the CSECT contains the executable code. This is followed by the program literals and the program's static data. Following the static data is the compiler-generated temporary storage. This is followed by the blocks defined in the program.

Each procedure's code consists of a prologue, a body, and an epilogue. The prologue and the epilogue perform the necessary operations to manipulate the stack of active procedures, establish basic addressability, etc. The body contains the procedure's working code. JOVIAL/J3B procedure mechanism and the basic run-time system to support its implementation are described below.

4.1 The CSECT Name of the Compilation

When the assembly language object file is produced by the compiler, the CSECT name is formed by truncating to seven characters, if necessary, the name taken from the main program or the first external procedure, function or block defined in the compilation and prefixing the resulting string with the character #. This convention for CSECT names prevents any conflicts with other JOVIAL/J3B external names (procedures, functions, or blocks) since the first character of those names can never be #.

4.2 External Conventions

4.2.1 Entry. Upon entering a JOVIAL/J3B 370 procedure, the following conditions must exist:

- R15 contains the address of the entry point
- R14 contains the return address
- R13 contains the address of a doubleword-aligned 72-byte or greater stack frame supplied by the caller
- R12 contains (except for the main program) the address of the basic run-time support control section \$AEDLNKG
- R11 contains the address of the argument list. R11 is zero if the procedure is called with no arguments.

4.2.2 Argument List. The argument list must be word-aligned and contain one word for each argument. The right-most three bytes of each word contains the 24-bit address of an argument. A word in the argument list may be one of the following.

- The address of a simple item, array element, unpacked item reference, or temporary location.
- The address of the first element in an array.
- The address of the zeroeth entry of a table.

The left-most bit of each word is a flag bit. The flag bit is 0 for all argument-list words but the last, for which it is 1.

4.2.3 Return. Upon return from a JOVIAL/J3B 370 procedure the following conditions exist:

- R15 is zero
- R0 contains the function value for integer or bit valued functions; otherwise the contents of R0 are unpredictable
- FPR0 contains the function value for a single float or double float valued function; otherwise the contents of all floating point registers are unpredictable.

All other registers are restored to their value at procedure entry.

4.3 Procedure Implementation

4.3.1 Body. In the body of a JOVIAL/J3B 370 procedure, general registers are used as follows:

- R0 integer arithmetic
- R1 integer arithmetic; argument list pointer when calling a procedure
- R2-R9 base and index registers for data addressability
- R10 program addressability. R10 contains address of procedure entry point
- R11 contains argument pointer (i.e., content of R1 upon entry to procedure)
- R12 contains address of basic run-time support control section (\$AEDLNKG)
- R13 save area pointer; dynamic data primary addressability register. Points to save area for next procedure call
- R14 procedure call return register; scratch index
- R15 procedure call address register; local addressability.

The procedure prologue does all necessary operations to establish the above environment for the procedure body. Similarly, the epilogue restores the corresponding environment of the calling procedure. The primary program addressability register, R10, contains the address of the procedure entry point.

4.3.2 Prologue. The functions performed by the prologue are:

1. Saves all registers in the save area supplied by the caller.
2. Obtains a new save area, links it to the current save area, and makes the new save area current.
3. Loads the basic program addressability register.
4. Loads the global data addressability registers.

Since functions 1, 2, and 3 are essentially identical for all procedures, the code necessary to perform them occurs only once in the JOVIAL/J3B 370 run-time support control section \$AEDLNKG. To address the \$AEDLNKG routines, R12 normally contains the address of the CSECT at procedure entry and is never modified by a JOVIAL/J3B 370 program. R12 is initialized with the address of \$AEDLNKG upon entering the main program.

Prologue function varies considerably with each procedure and is performed by in-line instructions in the procedure's executable code upon return from the \$AEDLNKG function. The compiler allocates up to four general registers for global data addressability. If needed, these registers are loaded during the prologue with the required address constants and remain set to those values throughout the execution of the procedure body. If additional registers are required for data addressability, general registers from the index register pool are temporarily assigned as needed.

4.3.3 Epilogue. The functions performed by the epilogue are essentially the reverse of the prologue functions 2 and 1. Because all epilogue functions are identical, they are performed by routines in the basic run-time support control section.

4.4 Basic Run-Time Support Routines

The basic run-time support routine for JOVIAL/J3B 370 programs perform the following functions:

For all procedures upon entry

- Save all registers in the calling program stack frame.
- Obtain a stack frame for the called procedure, link it to the old stack frame, and establish the new frame as the current stack frame.
- Load the primary program addressability register, R10 with the address of the called procedure.
- Load the argument list register, R11, from R1.

For all procedures upon exit:

- Release the current stack frame and establish the previous frame as the current frame.
- Set R0 (or FPR0) to the value of the procedure.
- Set R15 to zero.
- Restore all other registers from the stack frame.

In addition, the basic run-time support contains routines to perform integer-to-single and -double float and single- and double float-to-integer conversions; a run initialization routine, AEDINIT; run termination routines EXIT and ABEXIT; and a program interruption handler AEDCNTRL.

4.5 Stack Frame Management

All stack frames supplied by JOVIAL/J3B 370 procedures are dynamically obtained upon procedure entry and released upon exit. Stack frame management is performed by the JOVIAL/J3B 370 run-time support routines in control section \$AEDLNKG.

JOVIAL/J3B 370 procedures require 72-byte save areas.

Initially a stack frame pool with space for 20 stack frames is allocated. (The number 20 has been empirically determined and may be changed by an assembly parameter.) When this space is exhausted (i.e., more than 20 nested procedure calls take place) additional space for the pool is obtained from the supervisor in multiples of 2048 bytes and given to JOVIAL/J3B 370 procedures in appropriate sizes.

The stack frame management routines keep track of all the storage obtained from the supervisor by them. This storage is released upon job termination, signified by a normal return from the main program or by a call to procedure EXIT.

Figure 2-2 shows the use of each word in a stack frame. Word 0 is a control word used by the stack frame management routine. The leftmost byte of word 0 contains indicators identifying the type of stack frame. The rightmost three bytes contain the number of bytes in the stack frame.

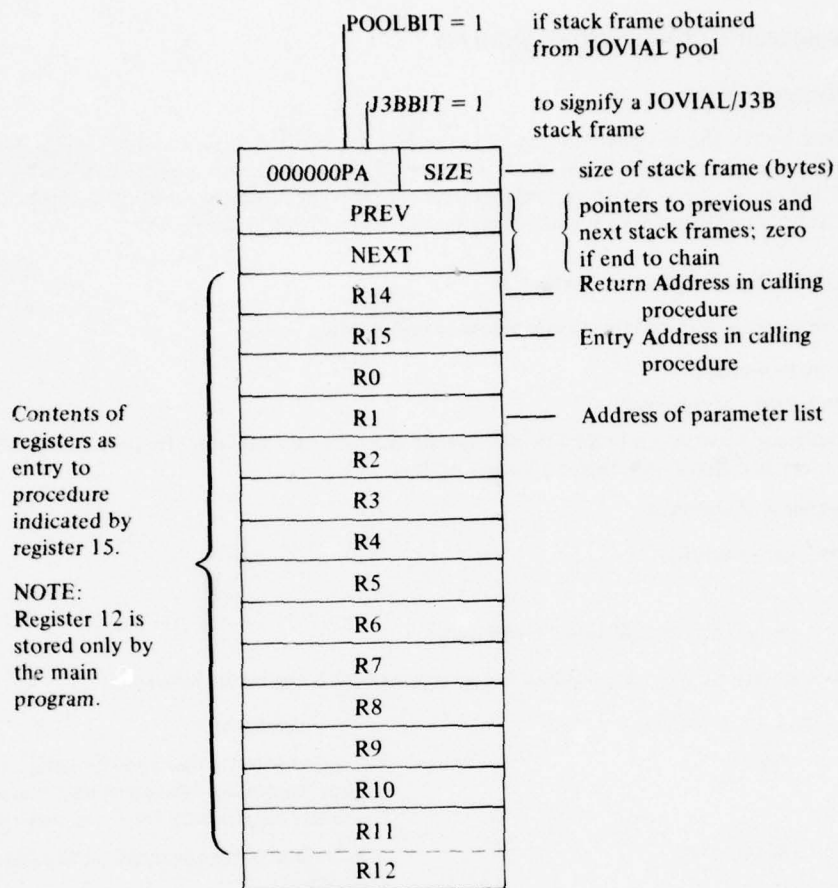


Fig.2-2 JOVIAL/J3B 370 stack frame layout

CHAPTER III

COMPILER ERROR MESSAGES

1. ERROR MESSAGE CONVENTIONS AND LIST

1.1 Error Message Printout

Errors detected by the compiler are printed in the first line available in the Source Listing. When no errors are detected, no messages are printed. When errors are detected, the compiler attempts to catch as many subsequent errors as practicable before stopping. However, certain errors may prevent continuation of the compilation process. When compilation is abnormally terminated, an Environment Listing will not be produced.

1.2 Individual Message Format and Content

Each error message contains three main pieces of information:

1. An error number.
2. The severity of the error.
3. Additional information to help further specify the cause of the error. This additional information is optional and may not appear in all error messages printed.

The printout is of the form

→ xx. yy severity

or

→ xx. yy severity:additional information

where xx. yy is an error number as described below and severity is one of the following:

<i>severity keyword</i>	<i>meaning</i>
NOTE	Not an error but rather an information message such as the symbol where normal processing is resumed after recovery from a syntax error.
WARNING	A user error which the compiler can correct reasonably.
ERROR	A user error which the compiler probably cannot correct.
SERIOUS.ERROR	A user error which the compiler definitely cannot correct.
SYSTEM.ERROR	A program error detected by a logical inconsistency in the compiler internal state.
SERIOUS.SYSTEM.ERROR	Same as SYSTEM.ERROR, but processing is terminated.

The additional information in some cases identifies more precisely the cause of the error. It may contain, for instance, the specific undeclared name which prompted the error message.

1.3 Error Message List

This section contains a list of compiler error messages. For easy reference, they are ordered according to the number printed with the error message. No attempt has been made to provide detailed information for error messages with severity of SYSTEM.ERROR or SERIOUS.SYSTEM.ERROR.

<i>Error No.</i>	<i>Field</i>	<i>Description</i>
1	1-30	System error
2	1-30	System error
3	1-11	System error
3	12	Program has too many (>20,000) significant characters
3	13	SS could not find primary J3B source file
3	14	SS could not find compool file
3	15-30	Error message missing
4	1-30	System error
5	1-6	System error
5	7	Double floating literal is too small
5	8	Single floating literal is too small
5	9	Single floating literal is too large
5	10	Double floating literal is too large
5	11	Too many hexadecimal characters
5	12	Integer literal is too large
5	13	Double floating literal has too much precision
5	14	Single floating literal has too much precision
5	15	A comment is terminated with a semicolon instead of a double quote
5	16-18	System error
5	19	Circular use of a defined name
5	20	System error
5	21-30	Error message missing
6	1-2	System error
6	3	S constant is too large in an item declaration
6	4	U constant is negative in an item declaration
6	5	U constant is too large in an item declaration
6	6	B constant is too large in an item declaration
6	7	C constant is too large in an item declaration
6	8	A name is multiply defined
6	9	Number of bits for an S item is greater than 31
6	10	Number of bits for a U item is greater than 31
6	11	Number of bits is missing for a U item
6	12	Number of bits missing for an S item
6	13	Number of bits is less than 1 for the S or U item
6	14	Number of bits is greater than 32 for a B item
6	15	Number of characters for a C item is greater than 132
6	16	Number of bits for a B item is less than 1
6	17	Number of characters for a C item is less than 1
6	18	Missing name field
6	19-23	Error message missing
6	24-25	System error
6	26	Array dimension(s) negative
6	27	Array dimension(s) zero
6	28	Array is too large (more than 32K)
6	29	System error
6	30	The lower bound of a table is greater than the upper bound
6	31	The upper bound of a table is less than 0
6	32-33	System error
6	34	Error message missing
6	35	The repetition constant is not an integer
6	36	System error
6	37	The starting bit for a table item is less than 0
6	38	The entry word for a table item is less than 0
6	39	The starting bit for a table item of type B or U is too big
6	40	The starting bit for a table item of type F or D is not equal to 0
6	41	The sum of starting bit + number of bits for a table item is greater than 31
6	42	The starting bit modulo 8 is not equal to 0 for a table item of type C
6	43	An item of type D appears in a parallel table
6	44	Initialization constant does not match the declared type
6	45	Initialization constant is too large for a table item of type S
6	46	Initialization constant is negative for a table item of type U
6	47	Initialization constant is too large for a table item of type U
6	48	System error
6	49	Initialization constant is too large for a table item of type B

*Error
No.*

Field

Description

6	50	Initialization constant is too large for a table item of type C
6	51	System error
6	52	Repetition constant is less than 1
6	53	System error
6	54	Words per entry is less than 1 in a table declaration
6	55-56	System error
6	57	Starting bit + 8 number of characters is greater than 32 for a type C item in a parallel table
6	58	Starting bit + number of bits is greater than 32 for a table item of type B or U
6	59	System error
6	60	The entry word is greater than words per entry - 1 for a table item
6	61	A table is too large (greater than 32K)
6	62	No match between an initialization constant and the declared type for either an array or a table item
6	63	Multiple definition of an item within a block
6	64-65	System error
6	66	A name which appears in a block definition has not previously appeared in a block declaration
6	67	Multiple initialization of the same item
6	68-69	System error
6	70	Multiple definition of an item within a procedure or function
6	71	Words per entry is less than 1 in a table declaration within a procedure of function
6	72	The lower bound of a table declared within a procedure or function is greater than its upper bound
6	73-75	System error
6	76	Multiple definition of a procedure
6	77	An item used within a procedure or function was not declared to be a formal parameter
6	78	The declaration is missing for an item which appears in a formal parameter description
6	79	The semantic type of a function is inconsistent between its definition and declaration
6	80	A formal parameter appears in a procedure or function declaration which has not been declared in the corresponding definition
6	81	The attributes of a formal parameter in a procedure or function declaration are inconsistent with the attributes in the corresponding definition
6	82	A constant of type B which is declared within a block is too large
6	83	Insufficient amount of initialization for a table or an array
6	84	Too much initialization for a table or array
6	85	An item declared within a parallel table overlaps another item
6	86	An item declared within a scalar or parallel table overlaps another
6	87	A table item of semantic type D is not aligned on an even word boundary
6	88-99	Error message missing
6	100-102	System error
6	103	Exponent overflow
6	104	Exponent underflow
6	105	Significance lost
6	106	Floating point division by zero
6	107	Integer overflow
6	108	Integer division by zero
6	109	System error
6	110	Overlay declaration appears outside of the block in which the first element belongs or the overlay declaration specifies an element which does not appear within the block
6	111	An element appears in more than one overlay declaration
6	112	Circular overlay constraint
6	113	Elements in an overlay declaration are not all in the same owning block
6	114-117	System error
6	118-123	Error message missing
6	124-199	System error
6	200	Storage requested for variable which has already been assigned storage
6	201	Storage requested for constrained variable which has already been assigned storage
6	202	Variable of incorrect type passed as argument to storage allocation
6	203	J3B program requires too much storage (>32K) as a result of a variable outside a block
6	204	J3B program requires too much storage (>32K) as a result of a variable within a block
6	205-209	Error message missing
6	210	A double precision item, array, or table has been constrained to a variable which is not aligned on an even word boundary

<i>Error No.</i>	<i>Field</i>	<i>Description</i>
6	211	A double precision item was aligned at a double word boundary causing one word of storage to be lost
6	212	A double precision array was aligned at a double word boundary causing one word of storage to be lost
6	213	A double precision table was aligned at a double word boundary causing one word of storage to be lost
6	214-216	Error message missing
7	1	System error
7	2	Syntax error
7	3	System error
7	4	Syntax restart. This indicates successful error recovery following error message 7.2
7	5-30	Error message missing
8	1-15	System error
8	16	Table name used in a nent functional modifier outside of the procedure or function of which the table name is a formal parameter
8	17-20	System error
8	21	Multiple reference of statement name: the statement name is defined in a procedure or function, but has been previously referenced in the main program or another procedure or function
8	22-24	System error
8	25	Negative index value in an indexed branch statement
8	26	Index value in an indexed branch statement is greater than the number of statement names in the corresponding switch declaration
8	27	A return statement appears in the main program
8	28	System error
8	29	A statement name is referenced in two different procedures or functions prior to its definition
8	30	A statement name is referenced in a branch statement in a procedure or function different from the one in which it is defined
8	31	A switch name is referenced in two different procedures or functions prior to its definition
8	32	A switch name is referenced in a branch statement in a procedure or function different from the one in which it is defined
8	33	A switch name is defined in a procedure after it was previously referenced in a different procedure
8	34	System error
8	35	A switch declaration contains fewer statement names than the value specified by the index in an indexed branch statement previous reference for this switch name
8	36	A B item is referenced outside of the procedure or function of which it is a formal parameter
8	37	A C item is referenced outside of the procedure or function of which it is a formal parameter
8	38	A D item is referenced outside of the procedure or function of which it is a formal parameter
8	39	An F item is referenced outside of the procedure or function of which it is a formal parameter
8	40	An S item is referenced outside of the procedure or function of which it is a formal parameter
8	41	A U item is referenced outside of the procedure or function of which it is a formal parameter
8	42	A B item is referenced outside of the procedure or function of which its owning scalar table is a formal parameter
8	43	A C item is referenced outside of the procedure or function of which its owning scalar table is a formal parameter
8	44	A D item is referenced outside of the procedure or function of which its owning scalar table is a formal parameter
8	45	An F item is referenced outside of the procedure or function of which its owning scalar table is a formal parameter
8	46	An S item is referenced outside of the procedure or function of which its owning scalar table is a formal parameter
8	47	A U item is referenced outside of the procedure or function of which its owning scalar table is a formal parameter
8	48	A B item is referenced outside of the procedure or function of which its owning one-dimensional table is a formal parameter

<i>Error No.</i>	<i>Field</i>	<i>Description</i>
8	49	A C item is referenced outside of the procedure or function of which its owning one-dimensional table is a formal parameter
8	50	A D item is referenced outside of the procedure or function of which its owning one-dimensional table is a formal parameter
8	51	An F item is referenced outside of the procedure or function of which its owning one-dimensional table is a formal parameter
8	52	An S item is referenced outside of the procedure or function of which its owning one-dimensional table is a formal parameter
8	53	A U item is referenced outside of the procedure or function of which its owning one-dimensional table is a formal parameter
8	54	A B table item is referenced with subscript less than the lower bound of its owning one-dimensional table
8	55	A C table item is referenced with subscript less than the lower bound of its owning one-dimensional table
8	56	A D table item is referenced with subscript less than the lower bound of its owning one-dimensional table
8	57	An F table item is referenced with subscript less than the lower bound of its owning one-dimensional table
8	58	An S table item is referenced with subscript less than the lower bound of its owning one-dimensional table
8	59	A U table item is referenced with subscript less than the lower bound of its owning one-dimensional table
8	60	A B table item is referenced with a subscript greater than the upper bound of its owning one-dimensional table
8	61	A C table item is referenced with a subscript greater than the upper bound of its owning one-dimensional table
8	62	A D table item is referenced with a subscript greater than the upper bound of its owning one-dimensional table
8	63	An F table item is referenced with a subscript greater than the upper bound of its owning one-dimensional table
8	64	An S table item is referenced with a subscript greater than the upper bound of its owning one-dimensional table
8	65	A U table item is referenced with a subscript greater than the upper bound of its owning one-dimensional table
8	66	A one-dimensional array of type B is referenced outside of the procedure or function of which it is a formal parameter
8	67	A one-dimensional array of type C is referenced outside of the procedure or function of which it is a formal parameter
8	68	A one-dimensional array of type D is referenced outside of the procedure or function of which it is a formal parameter
8	69	A one-dimensional array of type F is referenced outside of the procedure or function of which it is a formal parameter
8	70	A one-dimensional array of type S is referenced outside of the procedure or function of which it is a formal parameter
8	71	A one-dimensional array of type U is referenced outside of the procedure or function of which it is a formal parameter
8	72	A one-dimensional array of type B is referenced with a negative subscript
8	73	A one-dimensional array of type C is referenced with a negative subscript
8	74	A one-dimensional array of type D is referenced with a negative subscript
8	75	A one-dimensional array of type F is referenced with a negative subscript
8	76	A one-dimensional array of type S is referenced with a negative subscript
8	77	A one-dimensional array of type U is referenced with a negative subscript
8	78	A one-dimensional array of type B is referenced with a subscript greater than the declared dimension of the array
8	79	A one-dimensional array of type C is referenced with a subscript greater than the declared dimension of the array
8	80	A one-dimensional array of type D is referenced with a subscript greater than the declared dimension of the array
8	81	A one-dimensional array of type F is referenced with a subscript greater than the declared dimension of the array
8	82	A one-dimensional array of type S is referenced with a subscript greater than the declared dimension of the array
8	83	A one-dimensional array of type U is referenced with a subscript greater than the declared dimension of the array

<i>Error No.</i>	<i>Field</i>	<i>Description</i>
8	84	A two-dimensional array of type B is referenced outside of the procedure or function of which it is a formal parameter
8	85	A two-dimensional array of type C is referenced outside of the procedure or function of which it is a formal parameter
8	86	A two-dimensional array of type D is referenced outside of the procedure or function of which it is a formal parameter
8	87	A two-dimensional array of type F is referenced outside of the procedure or function of which it is a formal parameter
8	88	A two-dimensional array of type S is referenced outside of the procedure or function of which it is a formal parameter
8	89	A two-dimensional array of type U is referenced outside of the procedure or function of which it is a formal parameter
8	90	A two-dimensional array of type B is referenced with a negative first subscript
8	91	A two-dimensional array of type C is referenced with a negative first subscript
8	92	A two-dimensional array of type D is referenced with a negative first subscript
8	93	A two-dimensional array of type F is referenced with a negative first subscript
8	94	A two-dimensional array of type S is referenced with a negative first subscript
8	95	A two-dimensional array of type U is referenced with a negative first subscript
8	96	A two-dimensional array of type B is referenced with a negative second subscript
8	97	A two-dimensional array of type C is referenced with a negative second subscript
8	98	A two-dimensional array of type D is referenced with a negative second subscript
8	99	A two-dimensional array of type F is referenced with a negative second subscript
8	100	A two-dimensional array of type S is referenced with a negative second subscript
8	101	A two-dimensional array of type U is referenced with a negative second subscript
8	102	A two-dimensional array of type B is referenced with a first subscript greater than the declared first dimension of the array
8	103	A two-dimensional array of type C is referenced with a first subscript greater than the declared first dimension of the array
8	104	A two-dimensional array of type D is referenced with a first subscript greater than the declared first dimension of the array
8	105	A two-dimensional array of type F is referenced with a first subscript greater than the declared first dimension of the array
8	106	A two-dimensional array of type S is referenced with a first subscript greater than the declared first dimension of the array
8	107	A two-dimensional array of type U is referenced with a first subscript greater than the declared first dimension of the array
8	108	A two-dimensional array of type B is referenced with a second subscript greater than the declared second dimension of the array
8	109	A two-dimensional array of type C is referenced with a second subscript greater than the declared second dimension of the array
8	110	A two-dimensional array of type D is referenced with a second subscript greater than the declared second dimension of the array
8	111	A two-dimensional array of type F is referenced with a second subscript greater than the declared second dimension of the array
8	112	A two-dimensional array of type S is referenced with a second subscript greater than the declared second dimension of the array
8	113	A two-dimensional array of type U is referenced with a second subscript greater than the declared second dimension of the array
8	114	System error
8	115	Negative value for integer formula in shift functional modifier
8	116	Formal input parameter used as control variable in a for statement
8	117	No value assignment within B valued function
8	118	No value assignment within C valued function
8	119	No value assignment within D valued function
8	120	No value assignment within F valued function
8	121	No value assignment within S valued function
8	122	No value assignment within U valued function
8	123	System error
8	124	Recursive procedure call
8	125	Reference to undefined label
8	126	Reference to undefined switch name
8	127	Value assignment for B function occurs outside of function definition
8	128	Value assignment for C function occurs outside of function definition
8	129	Value assignment for D function occurs outside of function definition

Error

<i>No.</i>	<i>Field</i>	<i>Description</i>
8	130	Value assignment for F function occurs outside of function definition
8	131	Value assignment for S function occurs outside of function definition
8	132	Value assignment for U function occurs outside of function definition
8	133	User attempt to assign a negative constant to a U function
8	134	User attempt to assign to a U function a constant which is too large
8	135	User attempt to assign to an S function a constant which is too small
8	136	User attempt to assign to an S function a constant which is too large
8	137	User attempt to assign to an S function a signed variable which is too large
8	138	User attempt to assign to an S function a signed variable which is too small
8	139	User attempt to assign to an S function an unsigned variable which is too large
8	140	User attempt to assign to an S function an unsigned variable which is too small
8	141	User attempt to store into an input formal parameter of type B
8	142	User attempt to store into an input formal parameter of type C
8	143	User attempt to store into an input formal parameter of type D
8	144	User attempt to store into an input formal parameter of type F
8	145	User attempt to store into an input formal parameter of type S
8	146	User attempt to store into an input formal parameter of type U
8	147	Attempt to store non-zero constant into an entry variable
8	148	Attempt to store into an entry variable which is an input formal parameter
8	149	Assignment statement involves two entry variables whose owning tables have different number of words per entry
8	150	Attempt to store an entry variable into an entry variable which is a formal input parameter
8	151	Recursive call to a function of type C
8	152	Entry functional modifier has as its argument a scalar table which does not belong to the current procedure or function
8	153	Entry functional modifier has as its argument a one-dimensional table which does not belong to the current procedure or function
8	154	Subscript value less than lower bound for one-dimensional table in entry functional modifier
8	155	Subscript value greater than upper bound for one-dimensional table in entry functional modifier
8	156	Recursive call to a function of type D
8	157	Recursive call to a function of type F
8	158	Recursive call to a function of type S
8	159	Recursive call to a function of type U
8	160	Recursive call to a function of type B
8	161	Attempt to compare an entry functional modifier with an unsigned literal whose value is non-zero
8	162	Attempt to compare an entry functional modifier with an integer constant whose value is non-zero
8	163	Attempt to compare two entries of two tables which have differing words per entry
8	164	Constant specifying amount to shift for shiftr or shiftl functional modifier is too large
8	165	Procedure or function call with more actual input parameters than formal parameters given in the corresponding procedure or function declaration
8	166	Procedure or function call with more actual output parameters than formal parameters given in the corresponding procedure or function declaration
8	167	Procedure or function call has actual input parameter in position corresponding to formal input parameter in procedure or function declaration
8	168	Procedure or function call has actual output parameter in position corresponding to formal output parameter in procedure or function declaration
8	169	Procedure or function call has an actual input parameter which is a formula while the corresponding formal input parameter is an array or table
8	170	Procedure or function call has an actual input parameter which is a variable while the corresponding formal input parameter is an array or table
8	171	Procedure or function call has an actual output parameter which is unaligned and cannot be passed as an output parameter
8	172	Procedure or function has an actual output parameter which is a formal input parameter of the current procedure or function definition
8	173	Procedure or function call has no actual parameters although it was declared or defined with at least one formal parameter
8	174	Procedure or function call has fewer actual parameters than specified in its declaration or definition
8	175	Procedure or function call has a table as an actual parameter but the corresponding formal parameter in the declaration or definition is not a table

<i>Error No.</i>	<i>Field</i>	<i>Description</i>
8	176	Procedure or function call has a table as an actual parameter whose construction differs from the corresponding formal parameter
8	177	Procedure or function call has a one-dimensional table as an actual parameter whose upper bound differs from the corresponding formal parameter
8	178	Procedure or function call has a one-dimensional table as an actual parameter whose lower bound differs from the corresponding formal parameter
8	180	Procedure or function call has an array as an actual parameter but the corresponding formal parameter in the declaration or definition is not an array
8	181	Procedure or function call has an array as an actual parameter whose number of dimensions differs from those of the corresponding formal parameter
8	182	Procedure or function call has an array as an actual parameter whose first dimensions differs from that of the corresponding formal parameter
8	183	Procedure or function call has an array as an actual parameter whose second dimension differs from that of the corresponding formal parameter
8	184	Procedure or function call has a bit variable or array as an actual parameter which is not of a compatible semantic type with the corresponding formal parameter
8	185	Procedure or function call has a C type variable or array as an actual parameter which is not of a compatible semantic type with the corresponding formal parameter
8	186	Procedure or function call has a D type variable or array as an actual parameter which is not of a compatible semantic type with the corresponding formal parameter
8	187	Procedure or function call has a F type variable or array as an actual parameter which is not of a compatible semantic type with the corresponding formal parameter
8	188	Procedure or function call has a S type variable or array as an actual parameter which is not of a compatible semantic type with the corresponding formal parameter
8	189	Procedure or function call has a U type variable or array as an actual parameter which is not of a compatible semantic type with the corresponding formal parameter
8	190	Procedure or function call has a B type variable or array as an actual parameter which is not of compatible precision with the corresponding formal parameter
8	191	Procedure or function call has a C type variable or array as an actual parameter which is not of compatible precision with the corresponding formal parameter
8	192	Procedure or function call has a S type variable or array as an actual parameter which is not of compatible precision with the corresponding formal parameter
8	193	Procedure or function call has a U type variable or array as an actual parameter which is not of compatible precision with the corresponding formal parameter
8	194	The size field of an intr functional modifier is 0 or negative
8	195	The size field of an intr functional modifier is greater than the largest precision available for items of semantic type U
8	196	The size field of an intr functional modifier is greater than the number of bits available in the named variable field when considered as a bit string
8	197	Intr functional modifier has a negative index
8	198	Intr functional modifier has an index which is too large
8	199	The size field of a bit functional modifier is 0 or negative
8	200	The size field of a bit functional modifier is greater than the largest precision available for items of semantic type B
8	201	The size field of a bit functional modifier is greater than the number of bits available in the named variable field when considered as a bit string
8	202	A bit functional modifier has a negative index
8	203	A bit functional modifier has an index which is too large
8	204	The size field of a byte functional modifier is 0 or negative
8	205	The byte functional modifier size field is too large
8	206	Byte functional modifier has a negative index
8	207	The byte functional modifier index field is too large
8	208	Division by constant 0 was attempted
8	209-216	Error message missing
9	1	No initialized data or object code was generated because of syntax errors or null program
9	2-30	Error message missing
10	1-2	System error
10	3	The size of the run-time stack frame to be requested by the SKC object program will exceed the 256 word area addressable from the single register available for the base of the stack frame
10	4-8	System error
10	9	Too many (more than 10) actual parameters in an argument list
10	10	Too many (more than 10) formal parameters in an argument list

H-60

*Error
No.*

Field

Description

10

11

No initialized data or object code was generated because the source program contains only declarations

10

12-30

Error message missing

ANNEX I

LTR

MAXIRIS REAL-TIME LANGUAGE

FOREWORD

The MAXIRIS language was written to facilitate programming and development of real time systems. It is composed of a high level language based on ALGOL, to which a set of commands especially suited to real time, has been added. These commands allow communicating with the computer's multitasking monitor.

The specifications of this language allow efficient implementation on any computer manipulating values on 1, 2 or 4 bytes. They are, therefore, particularly valid for all 16-bit mini-computers.

Certain computers or certain monitor versions will require extra add-ons or restrictions. These will be described in the documentation associated with the implementation of the MAXIRIS language on these computers.

CONTENTS

	Page
1. INTRODUCTION	I-5
1.1 Purpose of this Document	I-5
1.2 Characteristics of the MAXIRIS Language	I-5
2. DESCRIPTION OF THE MAXIRIS LANGUAGE	I-5
2.1 MAXIRIS Program	I-5
2.2 Data Article	I-5
2.3 Expressions	I-9
2.4 Statements	I-10
2.5 Procedure Article	I-12
2.6 Standard Functions and Procedures	I-13
2.7 Complementary Definitions of the MAXIRIS Language	I-13
3. REAL TIME ASPECT	I-14
3.1 Basic Notions	I-14
3.2 Monitor Declarations	I-15
3.3 Monitor Data Expressions	I-15
3.4 Monitor Statements	I-17
3.5 Process Article	I-17
4. LINKS WITH THE OUTSIDE	I-21
4.1 Inputs-Outputs	I-21
4.2 Interrupts	I-22
APPENDIX – Glossary	I-23

1. INTRODUCTION

1.1. PURPOSE OF THIS DOCUMENT

This document describes the BASIC MAXIRIS language and its main characteristics and possibilities.

It is neither a programmer's manual nor a document describing the **implementation** of the language on a specific computer. These are given in specialized documentation.

As far as the monitor statements and the inputs/outputs are concerned, this document describes their general syntax as well as the principle of their effect. A detailed semantic description may only be made in connection with the multitasking monitor of the computer under consideration.

1.2. CHARACTERISTICS OF THE MAXIRIS LANGUAGE

MAXIRIS is a REAL TIME oriented high-level language. It differs from other programming languages in the two following essential points:

- It allows describing elements of programs working "in parallel", the execution of which may be interrupted to the advantage of other programs by waiting for an event or a resource or by the generation of a task.
- It offers the possibility of using structured data of a transitory nature.

Like ALGOL, this language has a block structure. The variables are declared at the head of the block and their scopes are those of the block and of the blocks inside this block. The same types of statements as those found in ALGOL are also found here:

- Assignment statements,
- Branch statements,
- Procedure calls,
- Loops,
- Conditional branches and statements.

On the other hand, the set of variable types available in this language is much wider than in ALGOL. Besides the conventional types, INTEGER, REAL, BOOLEAN, STRING, LABEL, the following are also found: LOGICAL, REFERENCE, QUALITY, RESOURCE, EVENT.

Most of these data may be found in a program under various forms:

- a constant,
- a variable,
- an n-dimensional array item,
- a set element field,
- a structure field.

Data of the same type may be combined by means of operators to form more complicated expressions which, in turn, may be combined.

The expressions are used in calculating values and it is the phrases (*or statements*) which manipulate these values.

The following possibilities are offered:

- to group several declarations having a generic name to form a **data article**.
- to group several statements having a generic name to form a **procedure article** whose call is an immediately executable statement.
- to group several statements having a generic name to form a **procedure article** whose call is a statement which may have deferred execution. It is the generation of a task running in parallel with the others. Each call to the process becomes a **task** which must be initiated by the monitor.

A program is presented as a succession of articles. Lastly, the language contains a certain number of ancillary possibilities to make it easier to use. For instance:

- implicit procedures of current use,
- the possibility of writing in assembly language within a program,
- the possibility of making macro-definitions,
- the possibility including comments.

2. DESCRIPTION OF THE MAXIRIS LANGUAGE

2.1. MAXIRIS PROGRAM

A MAXIRIS program is a succession of articles:

- data articles,
- procedure articles,
- process articles.

2.2. DATA ARTICLE

A data article contains declarations whose SCOPE is extended to the whole system if the article is GLOBAL

and associated to the SYSTEM DATA article, or, on the contrary, limited to the PROCESS or INTERRUPT PROCEDURE article to which it is associated.

Two types of data articles exist:

- ARTICLE DATA for the declaration of basic data:
 - arithmetic,
 - logical,
 - character strings,
 - quality,
 - reference,
 - boolean.
- ARTICLE SYSTEM DATA for the declaration of data concerning the monitor especially:

- events,
- resources,
- peripheral equipments,
- location modalities for the various articles,
- parameters necessary for monitor generation.

2.2.1. Basic Data declarations

A data is a piece of information stored in the memory, which may be used and/or modified by a program. In a MAXIRIS program, the basic data may appear in the following forms:

- constants,
- variables, subscripted or not,
- n-dimensional arrays of all types,
- structures (*association of LOGICAL, INTEGER, FIXED, REAL, STRING, BOOLEAN type fields*),
- tables of structures,
- bit strings for LOGICAL type information,
- character strings.

2.2.1.1. DECLARATION OF ARITHMETIC DATA

Arithmetic data are used in describing magnitudes (*length, rate, weight...*).

The arithmetic data are magnitudes represented by three types of numbers:

- INTEGER,
- FIXED,
- REAL.

An integer type number may be represented:

- on 1 byte : BYTE INTEGER,
- on 2 bytes : SHORT INTEGER,
- on 4 bytes : INTEGER.

A fixed type number corresponds to fixed point computer processing. A scale factor is associated with it. It may be represented:

- on 2 bytes : SHORT FIXED,
- on 4 bytes : FIXED.

A real type number corresponds to floating point computer processing. It is represented on 4 bytes.

These magnitudes may be initialized when they are declared.

Examples:

- INTEGER A, B : 45, C, D : 64, E, F;
6 integer variables (*each represented on 4 bytes*) are declared, 2 of which are initialized (B at 45, D at 64).
- SHORT FIXED 5 FX : -5.71, FY;
FX and FY are declared fixed point variables (*each represented on 2 bytes*), with a scale factor equal to 5, and FX is initialized at -5.71.

2.2.1.2. DECLARATION OF LOGICAL DATA

The logical data represent bit strings whose length may be:

- 1 byte : BYTE LOGICAL,
- 2 bytes : SHORT LOGICAL,
- 4 bytes : LOGICAL.

The initialization of bit strings may be expressed:

in binary : BIN,
in hexadecimal : HEX,
in octal : OCT.

Examples:

BYTE LOGICAL PAT, PIT : BIN'00011011';

Two bit strings PAT and PIT are declared. PIT is initialized in binary at 00011011. PAT and PIT are two bit strings with a maximum length of one byte. If a bit string name is followed by two arithmetic expressions between parentheses (A, B), it means that we wish to designate the portion of bit string beginning at the bit of portion a and extending over b bits, a and b being the results of the respective evaluation of A and B. The arithmetic expressions are called "STRING EXPRESSIONS".

Example:

PIT (5, 2);
PAT (3, 4);

Only bits 5 and 6 of PIT, and bits 3, 4, 5, 6 of PAT will be considered.

2.2.1.3. DECLARATION OF CHARACTER STRING

Character string data are alphanumerical data used in representing or manipulating texts.

The strings are stored in the memory at the rate of one character per byte : they are characterized by their length which is an integer ≤ 254 .

Example:

STRING 80 CARD ;
STRING 4 COD : 'ALFA';

2 character strings CARD and COD are declared. CARD is an 80-character string. COD is a 4-character string and is initialized by the constant 'ALFA'.

2.2.1.4. DECLARATION OF QUALITY DATA

The quality data are used to represent the nature of a status.

A quality is a data which may take a limited number (≤ 256) of different values. Each "value" is a name or attribute with which is associated a numerical code that can take the values : 0, 1, 2, ... 255.

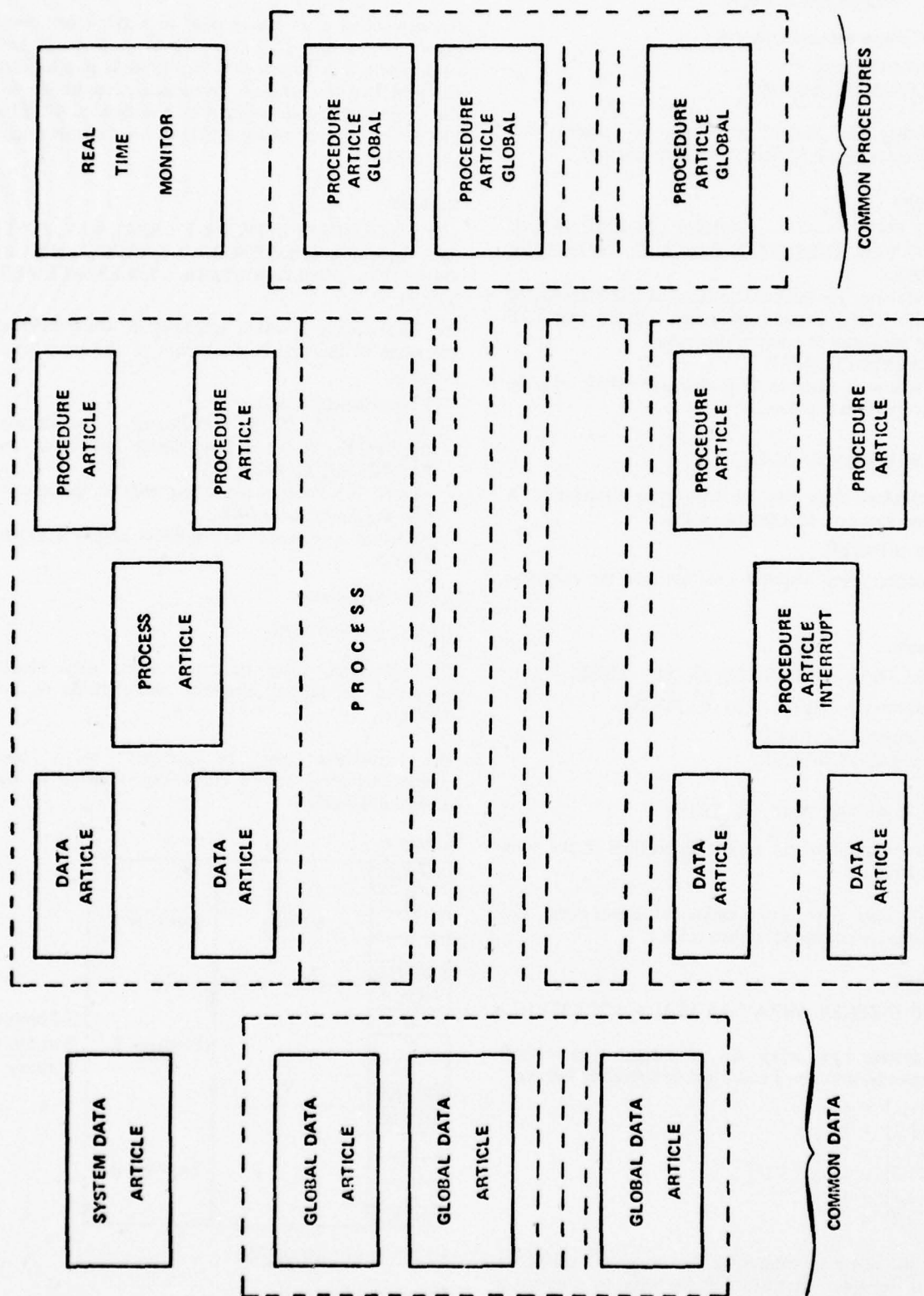
The quality data require one byte in memory.

Example:

QUALITY (RED 1, GREEN 2, BLUE 3,) COLOR,
C : BLUE;

Two qualities are declared, COLOR and C which may take, as value, one of the three following attributes:

RED (code 1),
GREEN (code 2),



General Structure of a Real Time Program

BLUE (code 3).

C is initialized at BLUE, thus at 3.

2.2.1.5. REFERENCE DATA

The reference data are pointers:

- to STATIC data,
- to DYNAMIC data : set.

When they are declared, they may be assigned to a specific data (*and initialized if the data is static*).

Examples:

- SET PIST (-----); SHORT LOGICAL ARRAY [10] TAB; REFERENCE PIST A, B; REFERENCE TAB C;
3 reference variables are declared: A and B associated with the PIST set and C associated with the array TAB and initialized to point to this array.
- REFERENCE ANY D;
1 reference variable D is declared which may be associated to any data.

2.2.1.6. BOOLEAN DATA

The Boolean data serve in expressing alternatives. A Boolean data may take the two values:

TRUE or FALSE.

A Boolean data requires one byte in the memory.

Example:

BOOLEAN B1, B2 : FALSE, B3, B4 : TRUE;

4 Booleans are declared : B1, B2, B3, B4.

B2 is initialized at FALSE.

B4 is initialized at TRUE.

2.2.1.7. ARRAY DECLARATIONS

All the basic data can appear in program in the form of an array.

Arrays may have any number of dimensions. The elements are numbered starting from 1.

Example:

BYTE INTEGER ARRAY AA [3,2] < 4;7; 3 TO 1;2>

We declare byte array AA an integer type with 2 dimensions, with size 3 and 2 and initialized as follows:

AA [1,1] = 4	
AA [2,1] = 7	
AA [3,1] = 1	
AA [1,2] = 1	} 3 TO 1
AA [2,2] = 1	
AA [3,2] = 2	

The six variables composing this array are called indexed variables. Actually, if we wish to identify a variable of array AA, Indexes must be used.

The name of a variable in this array will be, for instance : AA [1,2] : this is an indexed name. Subsequently, an indexed variable is a variable marked by an

indexed name. It has all the properties of a single variable.

2.2.2. Set declaration : SET

An element of a set corresponds to a data block composed of a string of variables. Each variable is called field. There may be any number of blocks in a set and these blocks are generally dynamically controlled by the monitor. They are pointed by a variable of REFERENCE type through which a field of a set element may be designated.

Example:

SET PIST (SHORT INTEGER T : REAL X, Y, Z, YP, ZP ; QUALITY (PASSENGER 0, CARGO 1, MILI 2) IDENTITY ; BOOLEAN STATE ; REFERENCE PIST NEXT) ;

We thus declare a set of airplanes in which PIST is the name of this set. Each element of the set is composed of:

- T aircraft detection time,
- X, Y, Z, YP, ZP the coordinates of the aircraft,
- IDENTITY is a quality whose attributes are PASSENGER, CARGO, MILI,
- STATE is a boolean indicating whether the aircraft is in the ground or in flight.
- NEXT is a reference to its direct neighbor in the PIST set.

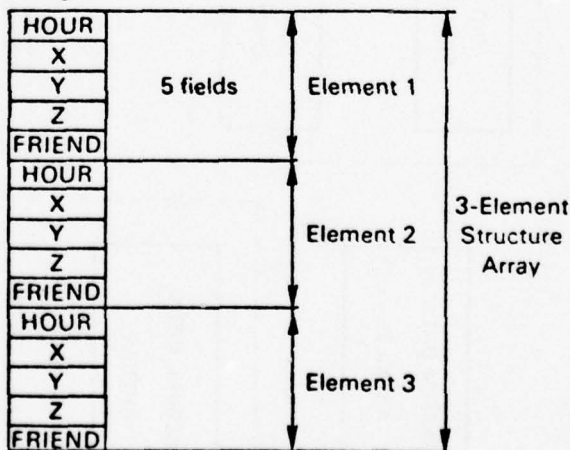
2.2.3. Structures

2.2.3.1. DEFINITION

The structures allow declaring static data blocks composed of several elements called fields of the structure.

The structures may be grouped in a one-dimensional array whose index may undergo an optimized processing.

Example:



2.2.3.2. DECLARATION

The general format is:

STRUCT {
 - ARRAY name [constant] (description of the element fields).
 - name (description of the element fields).

The preceding structure would be declared as follows:

STRUCT ARRAY PLANE [3] (SHORT INTEGER HOUR, X, Y, Z ; BOOLEAN FRIEND).

2.2.3.3. INITIALIZATION

The initializations are made following the declaration of element fields, in the order in which the fields are described, in a way identical to the initialization of single variable if the structure is not ARRAY, in a way identical to the initialization of an array if the structure is ARRAY.

2.2.3.4. INDEX

Its declaration has the format:

INDEX structure name {
- index name,
- ARRAY, index name,
dimension.

The use of the INDEX declaration is not mandatory since any integer may be used as an index of a structure array. However, the INDEX declaration allows the compiler to optimize access to the fields.

2.3. EXPRESSIONS

Expressions are the operands of the statements. According to their use in the statements, certain ones are used to designate a name, others to designate a value.

Example: A = B

A designates a name,
B designates a value.

An expression of value may be:

- a constant	} simple expressions
- a single name	
- an indexed name	
- a name with reference	
- with operators	} conditional expressions
- conditional	

2.3.1. Simple expressions

Any constant, any simple or indexed variable or variable with reference, any function call, may be used in a program as expressions.

The expressions may be linked by operators to generate a larger expression.

The arithmetic and logic expressions use their own operators.

A boolean expression uses its own operators, but they may link expressions of another type. The expressions of quality, character strings and reference do not use an operator.

2.3.1.1. ARITHMETIC OPERATORS

EXP : exponentiation

Example: A EXP B EXP C is read $(A^B)^C$

* : multiplication
/ : division
DIV : integer division
MOD : calculation of the remainder of the division of two integer operands (MOD is associated to DIV)

Example: A/B*C/10

+ : addition
- : subtraction

Example: A + B - C

2.3.1.2. LOGICAL OPERATORS

- shifts
SLL (n) logical shift to the left by n positions
SRL (n) logical shift to the right by n positions
SLC (n) circular shift to the left by n positions
SRC (n) circular shift to the right by n positions
SLA (n) arithmetic shift to the left by n positions
SRA (n) arithmetic shift to the right by n positions
- Complementation CPL
- Logical AND LAND
- Logical OR LOR
- Exclusive OR XOR

Moreover, a specific field of bits may be designated by means of the "STRING EXPRESSION" notation.

Example:

A (I,J) |K| designates bits I to I + J of the element of index K in array A.

- Comparison
GTR : >
GEQ : ≥
LEQ : ≤
LSS : <
EQL : =
NEQ : ≠
- Boolean operators
NOT
AND
OR

The operators have relative priorities. The standard use of parentheses allows specifying the exact order in which the operations will be executed in a complex expression.

Example:

$(A + B*(C + D)) \text{ GTR } ((A + B) / D) \text{ AND BETA}$

is a boolean expression which is true if:

$a + b(c + d) > \frac{a + b}{d}$ and β true

It is false in the opposite case.

2.3.2. Conditional expressions

The conditional expressions have the general format:
IF [boolean expression] THEN [expression X]

ELSE [expression Y]; where expression X and expression Y are the following:

- arithmetic,
- logical,
- boolean,
- character strings,
- reference,
- or quality.

Example of a conditional expression:

IF X EQL 5 THEN Y ELSE Z

If X, Y, Z have been declared as arithmetic variables, the conditional expression above is an arithmetic conditional expression. Like any expression, it acts as an operand in a statement. For instance:

A = IF X EQL 5 THEN Y ELSE Z

which is interpreted:

$X = 5 \Rightarrow A = Y$

$X \neq 5 \Rightarrow A = Z$

2.3.3. Designation expressions

The simple designation name is a name associated, by a declaration made inside a block, with the LABEL type. It constitutes the statement labels in the block in which it was declared. The label declaration has the format:

LABEL name

The designation expressions are used in GO TO statements.

2.4. STATEMENTS

Statements constitute the active part of the language. They define actions through operators. Distinction may be made between:

- simple statements whose operands are expressions,
- compound statements whose operands are statements or possibly expressions.

The labelled statement (*statement marked by a label*) may be simple or compound.

A label name is a name which, when declared in a block, was associated with the key word LABEL.

Examples:

- GO TO LABL1 is a simple statement
 - the operator is GO TO
 - the operand is LABL1 (*designation expression*)
- IF A GTR B THEN A = B ELSE GO TO LABL1 is a compound statement
 - the operator is IF ... THEN ... ELSE ...
 - the operands are:
 - A GTR B (*boolean expression*)
 - A = B (*statement*)
 - GO TO LABL1 (*statement*)
- LABL1 : A = B is a labelled statement.

2.4.1. Simple statements

The operands of a simple statement are made up of expressions only. A simple statement may be labelled. Among the simple statements are:

- assignment statement,
- designation statement,
- empty statement,
- monitor statement,
- dynamic data control statement,
- block.

2.4.1.1. ASSIGNMENT STATEMENT

This statement assigns the value of an expression to a name expression (*simple or conditional*).

Example:

A = X EXP 2 + X*Y + 2

is an arithmetic assignment statement.

A is an arithmetic name.

EXP 2 + X*Y + 2 is an arithmetic expression; by this statement, the value of the expression: $X^2 + XY + 2$ is assigned to A.

In the same way, there are assignment statements which are boolean, logical, character string, quality or reference.

2.4.1.2. DESIGNATION STATEMENTS

There are two types of designation statements:

- GO TO designation expression,
- RETURN.
- GO TO designation expression.

This statement causes a sequence change. The statement to be executed is the one marked by the label constituting the result of the evaluation of the expression following GO TO.

Examples:

- GO TO LABL1,
- GO TO IF A LSS C THEN LABL1 ELSE LABL2.
- RETURN

This statement causes a branch to the end of the block in which it is located.

Examples:

BEGIN A = B;
IF A GTR 5 THEN RETURN;
A = A + 5;
C = EXP 2;
END;

If A is > 5, branch to END, if not, we execute:

A = A + 5

and C = EXP 2 before branching to END.

2.4.1.3. EMPTY STATEMENT

This is equal to "nothing".

It is useful in the case of the iteration statement, since it allows varying an index until a condition is obtained.

Example:

FOR I = 1 + 1 WHILE A [I] GTR 10 DO;

In this example, we increase I by 1 until we find I such that $A[I] \leq 10$; this is the only action.

2.4.1.4. DYNAMIC DATA CONTROL STATEMENT

These statements assign the reference to data areas. There are two different cases:

- **Dynamic data area**
This assumes that the monitor controls a dynamic data area.

The statement NEW X IN Y WITH (...) allows:

- obtaining, from the dynamic data management system, a data block which may contain a set element designated by Y,
- to make reference X point to the block obtained,
- possibly, to initialize the fields of the set.

Example:

SET PIST (SHORT INTEGER X, Y, Z, VX, VY, VZ);
REFERENCE ANY A1;
NEW A1 IN PIST WITH [X1,X2,X3,V1,V2,V3]

A1 points to a data block of size greater than or equal to 12 bytes and initialized.

To Y (A1) corresponds the value X2.

The statement ERASE:

- turns back the block pointed at by the reference to the dynamic area control system.
- puts the reference back to NIL.

Example: ERASE A1

- **Static data area**

The phrase POINT X ON Y allows making a reference X point to a data Y of the program. This allows generating an address constant in the process of execution.

Example:

STRING 80 BUF1, BUF2;
STRUCTURE CB (SHORT LOGICAL FUNCTION : X'00F3', REFERENCE ANY PTBUF, SHORT INTEGER LONG : 80);
POINT PTBUF ON BUF1; ES [CB];
POINT PTBUF ON BUF2; ES [CB];

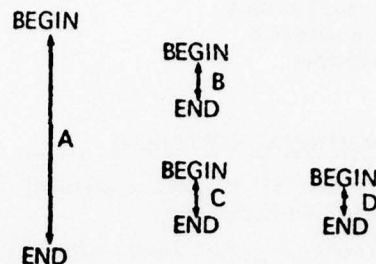
The structure CB is an input/output control block supplied to the external routine ES [], the two POINT statements allow making an input/output successively with BUF1 and then BUF2.

2.4.1.5. BLOCK

The block is a set which may always be used wherever a statement is authorized.

The block is a part of a program delimited by the words BEGIN and END.

The block groups several declarations and several statements. Any variable declared in a block may be used only inside this block. It is possible to imagine a more or less complex interleaving of blocks.

Example:

From the terminology point of view, we shall say that B, C, D are descendants of A, and vice versa, that A is their ascendant. C is also ascendant of D. A variable declared in A could be used in B, C and D. The reverse is not true.

Example:

IF A EQL B THEN
BEGIN C = D; E = F; END
ELSE J = K;

After THEN only one statement can be written, where two assignments are to be made. The problem is solved by creating a block containing these two assignments.

A descending block can use the following identifiers:

- data article identifiers,
- ascending block identifiers,
- identifiers belonging to the block.

An identifier already used in a data article of an ascending block may be redeclared in a descending block, the identifiers referring to a different type. It is the last definition which is attached to it as long as we do not go out of the block in which it was declared. Once out, it takes back the former significance.

Example:

```

BEGIN
  INTEGER A, B; LABEL L1, L2;
Integer A  A = X EXP 2; B = X
          BEGIN BOOLEAN A;
Boolean A  GO TO IF A THEN L1 ELSE L2
          END;
Integer A  L1 : B = A + B;
          GO TO L3;
Integer A  L2 : B = A + 2B;
          L3 :
          END
  
```

In this example, X is assumed already defined at a higher level. A is first an integer, then at its redeclaration, it is considered a boolean, then again becomes an integer at the output of the internal block. B remains integer throughout.

2.4.2. Compound statement

The compound statement defines action through an operator of which at least one of the operands is a statement.

Among the compound statements, there are:

- the conditional statement,
- the iteration statement,
- the case statement.

2.4.2.1. CONDITIONAL STATEMENT

- General format: IF [boolean expression] THEN [statement] ELSE [statement];
 - If the boolean expression has the value TRUE, the statement following THEN is executed and we go on to the next statement of the program;
 - If the boolean expression has the value FALSE, it is the statement following ELSE that is executed and we go on to the next statement of the program.
- Simplified format: IF [boolean expression] THEN [statement];
 - If the boolean expression has the value TRUE, the statement that follows THEN is executed, if not, we go immediately on to the next statement of the program.

Examples:

- IF A EQL B THEN
BEGIN C = D; E = F END
ELSE J = K;
 $A = B \Rightarrow C = D \text{ and } E = F$
 $A \neq B \Rightarrow J = K$
- IF X EQL 5 THEN Y = X EXP 2;
X = X + 1
 $X = 5 \Rightarrow Y = X^2$ then incrementation of X.
 $X \neq 5$, immediate incrementation of X.

2.4.2.2. ITERATION STATEMENT

The iteration statement allows automatically processing all the program loops.

- FOR-STEP format:
FOR [arithmetic assignment statement] STEP [arithmetic expression] UNTIL [arithmetic expression] DO [any statement];

Example: FOR X = 2 STEP 1 UNTIL 4 DO Y = X - 1

Different values are assigned to X by this statement. The first time X = 2. Then, we add the value of the STEP and so on until X reaches the value 4. For each value of X, the statement following DO is executed, i.e. Y = X - 1.

- FOR-WHILE format:
FOR [arithmetic assignment statement] WHILE [boolean expression] DO [any statement]

Example: FOR X = X - 1 WHILE X LEQ 4 DO Y = X - 1

After FOR we have the statement X = X - 1 for which we assign to X its present value increased by 1 unit. For each value of X, we execute the statement Y = X - 1 until X does not exceed 4.

2.4.2.3. CASE STATEMENT

This statement is used in programming switches.

General format: CASE [integer arithmetic expression] OF [block].

If the arithmetic expression equals n, the case statement allows executing, if it exists, the nth statement of the block.

The block output is automatic.

Example: CASE I OF
BEGIN

X = X + 12;
X = X + 21;
X = X + 30;
END

If: I = 1, statement X = X + 12 is executed,
If: I = 2, statement X = X + 21 is executed,
If: I = 3, statement X = X + 30 is executed.

After the chosen statement has been executed, there is automatically a jump to the end of the block. If I > 3, the case statement has no effect (*jump to the end of the block*).

2.5. PROCEDURE ARTICLE

A procedure article associates a name to a processing. Two types, of procedures exist:

- the procedure itself,
- the function which is a specific case of procedure.

2.5.1. Procedure

2.5.1.1. DECLARATION

When the same processing has to be carried out at several places in the program, it is preferable to group the statements of this processing and to associate a name to this group, the processing then takes place simply by referencing this name.

This group of statements (*sub-routine*) is called PROCEDURE in MAXIRIS. A procedure may:

- not have any formal parameters if the same processing is always to be carried out on the same data and for the same expressions.
- have a list of formal parameters if the same processing is to be carried out on different data and/or expressions. In this case, there will be the same number and same type of actual parameters as the formal parameters of the declaration.
Let us assume that we have to make calculations on complex numbers. It will be interesting to set up the sub-routine which calculates the real and imaginary parts R and X of the product of two complex numbers having, for real and imaginary parts, the quantities R1, X1 and R2, X2 respectively.

This sub-routine will be written as follows:

ARTICLE PROCEDURE PRODC (REAL VALUE


```

R1; REAL VALUE X1; REAL VALUE X2;
REAL R,X)
BEGIN
R = R1*R2 - X1*X2;
X = R1*R2 + R2*X1
END

```

As shown in the preceding example, the sub-routine is composed of a heading and a block. In the heading, besides the ARTICLE PROCEDURE, are found the name attributed to this sub-routine: PRODC and then a parameter declaration list.

2.5.1.2. CALL

The procedure call constitutes a statement of the language. The procedure call PRODC declared previously will be:

```
PRODC [I, J, K, L, RES1, RES2];
```

assuming that the parameters between [] were previously defined.

This sub-program has therefore allowed calculating the two quantities R and X symbolized by RES1 and RES2.

2.5.2. Function

2.5.2.1. DECLARATION

The function is a particular case of procedure: it is used to calculate only one quantity. The structure of such an article is equivalent to that of a procedure article (*heading + block*) with the two following variants:

- the type of function is specified in the heading.
- the block includes the phrase `RESULT = ...` which is the value of the function.

Say, for instance, that quantity $n!$ has to be calculated. The sub-routine will be written as follows:

```

ARTICLE INTEGER PROCEDURE FACT (N)
BEGIN INTEGER I; F:1;
FOR I = 1 STEP 1 UNTIL N
DO F = F*I;
RESULT = F;
END

```

2.5.2.2. CALL

The call for a function does not constitute a statement but an expression and will therefore be used directly as an operand in a statement.

A function call allows obtaining a value, of a determined type, as result of the application of a calculation rule; this rule having been specified upon declaration of the function.

The actual parameters are values which must replace the formal parameters in the function evaluation.

Example: If we wish to assign $10!$ to A, we shall write:

```
A = FACT [10]
```

2.5.3. Parameters

In a declaration of procedure (*or function*) may be found, after the name of the procedure (*or of the function*), a list of declarations between (). These are the formal parameters.

When the procedure is called (*or the function*), a list of expressions between [] may be found after the name of the procedure (*or the function*). These are the actual parameters. They will replace the formal parameters when the sub-routine is executed.

The number of current parameters is identical to the number of formal parameters. Their type is identical also.

Any variable may be a formal parameter. If we add the mention VALUE to a formal parameter, this means that we are interested in the value of the actual parameter that will replace it every time. If VALUE is not mentioned, we are interested in the address of the actual parameter.

2.6. STANDARD FUNCTIONS AND PROCEDURES

Standard functions and procedures exist in the operating system's library. They must be declared before use by a specific article of the following format:

ARTICLE type (*if it is a function*) FUNCTION name (*formal parameters*).

2.7. COMPLEMENTARY DEFINITIONS OF THE MAXIRIS LANGUAGE

2.7.1. Writing in assembly language

The writing in assembly language is reserved for:

- special problems requiring the use of the machine code,
- reasons of very stringent optimization of certain program sequences.

This writing is done by inserting the CODE directive into an article (*except for ARTICLE DATA*). It indicates that everything that follows up to the next ";" is assembly code and must be reproduced in extenso in the code generated.

2.7.2. Macro-generation

Its main objective is:

- a) to allow modifying key words,
- b) to reduce the writing of repetitive statements,
- c) to allow syntax evolution.

To this end, a macro-generation will be made at the source language level. This is applied right from the lexical analysis. In order not to degrade the compiler performances, the possibilities of macro-generation have been limited to simple substitutions of character strings.

Two possibilities are offered:

2.7.2.1. MACDEF

- A character string will be substituted for the name of the MACDEF.

This substitution is described by the declaration:

MACDEF name% string%

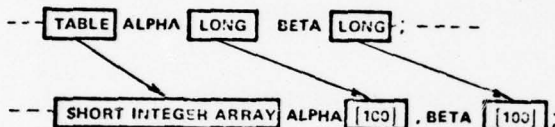
This declaration must precede its use. Its effect follows the same rules as the other declarations:

Every time "name" is encountered, it is replaced by "string".

Examples:

- **MACDEF TABLE% SHORT INTEGER ARRAY%**
MACDEF LONG% [100]%

allows simplifying the declaration of arrays having the same type and the same size.



- **MACDEF CHAMP1% (3.2)%**
MACDEF CHAMP2% (8.2)%

allows simplifying the referencing of:

--- **ALPHA CHAMP1 = BETA CHAMP2 [I]**

which will be changed in:

--- **ALPHA (3.2) = BETA (8.2) [I]**

2.7.2.2. MACPRO

MACPRO allows including parameters in the substitution. The syntax has the general format:

MACPRO macname input-prototype% output-prototype%

Input and output prototypes are composed of a succession of character strings without b, of separators, of key words "\$i" used to mark the parameters to be substituted.

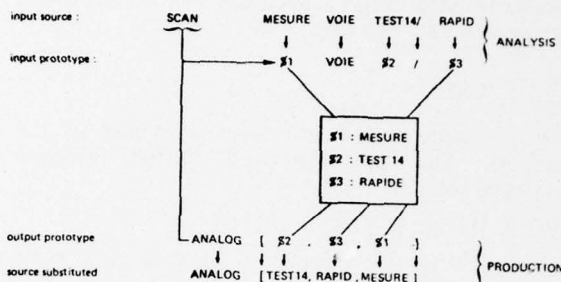
The macproduction is carried out as follows:

when "macname" is encountered in the source text, the input prototype is used to isolate the character strings corresponding to the parameters in the rest

of the source text. The latter are then inserted into the output prototype at the locations marked by the \$i and this result replaces the source text identified by the input prototype.

Example:

MACPRO SCAN_\$1_VOIE_\$2/\$3%
ANALOG [\$2,\$3,\$1]%



thus **ANALOG[TEST14, RAPIDE, MESURE];** will be substituted for **SCAN MESURE VOIE TEST14/RAPIDE;**

Remark:

- The blank separators (b) which are not significant of the input source text are eliminated.
- The MACPRO's must be declared before use. Their scope is the same as that of the identifiers.

2.7.3. Comments

The role of the comments is to document the programs. They do not generate any machine code. The comment statement is written:

COMMENT comment;

It may be found anywhere in a program.

2.7.4. Declaration of equivalence

The declaration of equivalence allows using a data with different types:

EQUIV single data name, single data name, ...

SHORT INTEGER IA;
SHORT LOGICAL IB;
EQUIV IA, IB

In this example IB (1.3) represents the 3 right-hand bits of the number IA.

3. REAL TIME ASPECT

3.1. BASIC NOTIONS

The notion of procedure is insufficient to solve certain programming problems of real time data

processing systems. The notion of process and task must be substituted for the notion of procedure.

Actually, a procedure execution takes place when the procedure is called whereas this is not the case for the processes. The execution of a process (Task) may be scheduled in time by means of certain

monitor commands handling the resources, the events and bringing in priorities. Such monitor commands may be used both to schedule a task in time when it is generated and to suspend its execution so as to pass control of the computer to other waiting tasks, and then to take back control of the interrupted task according to certain criteria.

The execution of a task generally consists of a sequence of active phases cut up by waiting phases which may correspond, for instance, to the waiting for an input/output, for an event, etc...

3.2. MONITOR DECLARATIONS

These declarations have the following common characters:

- they may be declared only in the form of variables (*and not arrays*),
- they are most often found in the monitor data article (ARTICLE SYSTEM DATA).

3.2.1. Event declaration

An event is an entity having two states:

- occurred,
- not occurred.

The tasks can wait until an event goes into occurred state.

Monitor statements (SETEV and RESETEV) allow fixing the status of an event.

The statement ACTIVATE combines the two preceding actions and carries out a "pulsed event". Only the tasks waiting when ACTIVATE is executed could re-start their execution.

An event cannot be declared in a procedure article. On the other hand, it may be declared inside a block of a process article. Its existence as an event name stops at the output of block in which it is declared.

An event may run as an actual parameter for a process call and therefore be a possible formal parameter in a process.

An event may be initialized. This allows ensuring the interface with monitors in which the events are represented by numbers.

Example: EVENT A, B, C : 3;

3.2.2. Resource declaration

A resource is an entity which may be reserved or released (*see statement, paragraph 2.4.1.4*).

A resource is also characterized by its maximum number of users. A resource has two states:

It may be *free*, in which case it may be reserved, and the number of supplementary possible users decreased by 1. By releasing it, this number is increased by 1.

When the number of supplementary users is zero, the resource is **busy** and anyone who wishes to reserve it must wait until one of the current users releases it. An unused resource reservation therefore suspends the task requesting it until its release.

The resources must be declared in the ARTICLE SYSTEM DATA.

A resource may be run in an actual parameter for a process call and, therefore, be a formal parameter in a process.

A resource may be initialized. This operation allows ensuring the interface with a monitor managing resources itself.

Example:

RESOURCE ALPHA : 4 USERS 6 declares the resource ALPHA with number 4 and parallelism 6.

3.3. MONITOR DATA EXPRESSIONS

The events and resources appear in simple form only since they cannot be declared in the form of arrays or set elements.

The resources are referenced only in monitor statements in which they enter as independent arguments:

Example: RESERVE ALPHA

The events will be referenced only in the monitor statements and they can intervene in the form of an expression combining events and delays.

Delay expressions

These expressions define a time interval. The unit used may be implicit (*case of only one unit possible or the last unit set by a TIME UNIT statement*) or explicit. The delay expression is of the SHORT INTEGER type:

Example:

ALPHA
(5 + ALPHA)
ALPHA UNIT GROS
6 UNIT 4

Date expression

This expression uses a date which is represented in memory by a structure.

Example:

STRUCTURE DATEX (SHORT INTEGER
YEARX:1980, DAYX:213, HOURX:8, TOPX:10000)

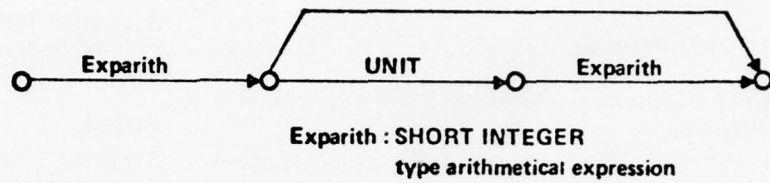
----- AT DATEX -----
HOURX = 20, ----- AT DATEX -----

Event expression

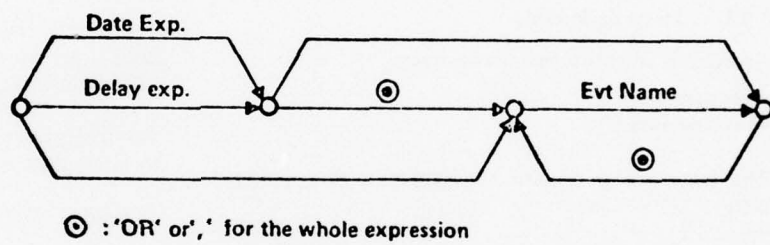
These expressions combine:

- delay expressions,

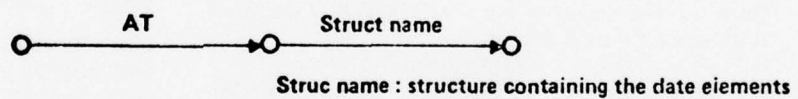
DELAY EXPRESSION



EVENT EXPRESSION



DATE EXPRESSION



MONITOR DATA EXPRESSIONS

- hour expressions,
- event references.

Example:

INTEGER START;
SHORT INTEGER LAG;
EVENT RUN, STOP;

LAG UNIT 3 or RUN
START or RUN

3.4. MONITOR STATEMENTS

The monitor statements allow the synchronization of task executions (*multitasking*). They therefore bring in events, times or resources.

Most of the time they lead to modifying the status of the tasks. This results in a new eligible task waiting queue. The scheduling algorithm selects the eligible task with the highest priority in order to give it control. In this case, the task executing a monitor statement calling upon the scheduling algorithm momentarily loses control.

This recourse to the scheduling algorithm is indicated in the language by the keyword MONITOR.

The actions defined by the monitor statements may have an effect which is immediate, deferred or periodical:

- deferred effect:
 - time expression : XX UNIT XX
 - hour expression : AT XXXX
- periodical effect:
 - EVERY time expression.

Control of the time unit

TIMEUNIT : allows choosing the implicit time unit.

Events and waits

SETEV sets the event in "OCCURRED" status.
RESETEV sets the event in "NOT OCCURRED" status.
ACTIVATE carries out the occurrence of a "pulse" event (*equivalent to SET + RESET*).
TESTEV tests the status of an event.
WAIT waiting for an event expression with call to the scheduling algorithm.
DELAY waiting for a delay without scheduling algorithm call.

Resources

RESERVE reserves a resource
FREE releases a resource.
STATE test of a resource with possible reservation if the resource is free.

Miscellaneous

MONITOR call to scheduling algorithm.

TASK SHORT INTEGER type function corresponding to the logical number of the task in process (*No. assigned by the monitor*).
DATE gives the date in the form of a structure or an array.
TIME INTEGER type function giving the "time".

Remark:

The preceding statements allow describing in detail the action of the multi-tasking monitor. In practice, a monitor does not allow obtaining all the combinations possible. In particular:

- *It implicitly imposes, for each statement, the recourse or not to the scheduling algorithm. This recourse is to be analyzed at the level of each application.*
- *It allows only a limited number of combinations for event expressions.*

This results in a list of restrictions and could be checked by the compiler.

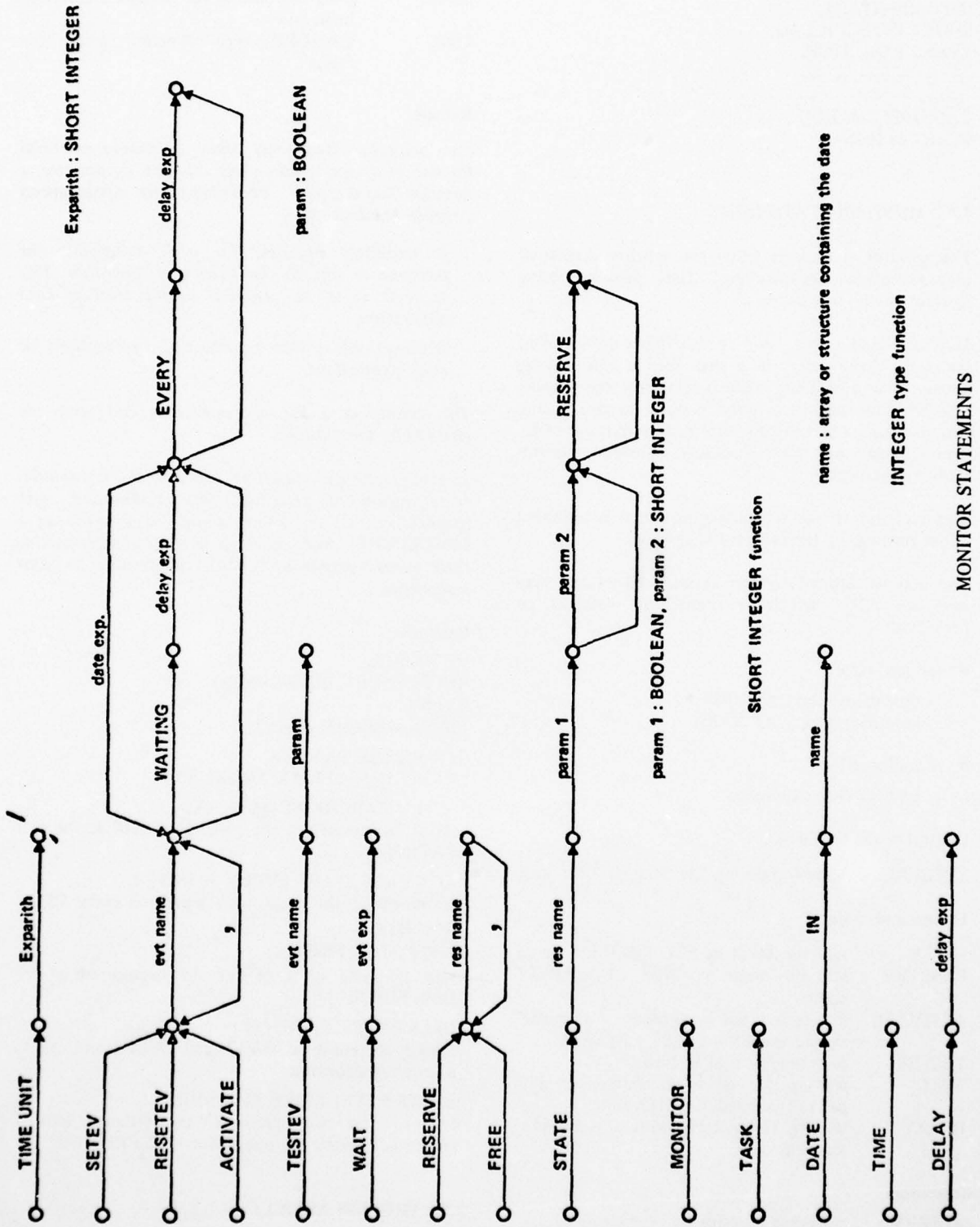
Inversely, certain monitors offer extra possibilities (modification of priorities, time suppression, task suppression, etc...). These could be entered as a PROCEDURE. The macro-generator allows making their syntax analogous to that of standard monitor statements.

Examples:

RESOURCE:
BUFF1 USER1, BUFF2 USER1;
EVENT:
FINES, MARCHE, START;
STRUCTURE DATE XX:
(SHORT INTEGER AX, JX, HX, TX)
*SETEV MARCHE AT DATE XX;
set at "occurred" of the event MARCHE at the date DATE XX.
*ACTIVATE START EVERY 50 UNIT 2
activation of the event with repetition every 50 units of type 2.
*WAIT 20 OU FINES
wait 20 time units or for the occurrence of the event FINES.
*FREE BUFF1, BUFF2;
release of resources BUFF1 and BUFF2 and call to scheduling algorithm.
*STATE BUFF1 LIBRE RESERVE;
test of the occupation of the resource BUFF1 (*status in LIBRE*) and reservation if BUFF1 is free.

3.5. PROCESS ARTICLE**3.5.1. Declaration**

A process article associates a name to a processing block. The mention of the process name in an article will generate a task which is not necessarily executed immediately.



Its declaration is composed of a heading and of a block like that of the process article:

```
ARTICLE PROCESS name [formal parameters]
BEGIN ----- END ;
```

3.5.2. Process call

The process call is a task generation. It arrives, at least in the beginning, like the procedure call. Then come certain other data:

- CLOSED : The calling process loses control until the task it has just generated by a closed process call is fully terminated.
- OPEN : The process calls another process in open mode.
- PRIORITY : < arithmetic expression > gives a certain priority to the task currently generated by the call, with respect to the tasks already generated or to come. If PRIORITY is absent, it is:
 - either implicit (associated in a fixed way with the process)
 - or that of the caller.

Then comes an optional information which allows a deferred execution while waiting for an event expression to be satisfied. This information has the form WAITING <event expression >.

The task generated is then put into waiting for an event in the same way as when, in the execution of a task, WAIT <event expression > is found (paragraph 3.4). The task will become eligible for the first time only when the <event expression > is satisfied. It could be periodically reactivated (argument EVERY).

Example:

Take the following process:

```
ARTICLE PROCESS ALPHA [INTEGER VALUE X,
INTEGER VALUE Y, INTEGER VALUE Z]
BEGIN
-----
-----
END
```

It could be called as follows:

```
ALPHA [I, J, K] CLOSED PRIORITY 5;
```

```
-----
IF REPET THEN
```

```
  - ALPHA [I, J, K] OPEN PRIORITY 4 EVERY T
    UNIT 2 ELSE
    ALPHA [I, J, K] OPEN PRIORITY 3 WAITING 4
    UNIT 2 OR READY.
```

- at the time of the first call, the caller waits for the end of the execution of process ALPHA (closed call).

- at the time of the second call if:

- REPET is true, the process ALPHA is periodical-

ly initiated, we do not wait for the end of the execution of ALPHA.

- REPET is false, ALPHA is initiated either after a time delay or upon the occurrence of the ready event.

Remark:

Depending upon the monitors:

- the scheduling algorithm will or will not be called as soon as the new task is generated.
- the execution of tasks will or will not be reentrant,
- the generation of tasks will or will not be put into queue.

In case of no-re-entrance and no queuing, the monitor itself will put the caller into waiting for the end of execution of the called task.

Every time the re-entrance ensured by the monitor is insufficient, resources must be used to avoid any risk of losing data.

Example:

No-re-entrant tasks, argument field in common data.

```
RESOURCE RESP1 USERS 1; COMMENT RESOURCE
ASSOCIATED TO PROCESS PROC1 AND HIS ARGU-
MENTS;
```

*calling:

```
-----; RESERVE RESP1; fill in parameters; PROC1
OPEN PRIORITY 3; -----
```

*called:

```
ARTICLE PROCESS PROC1
BEGIN
```

```
-----
processing
```

```
FREE RESP1;
END;
```

Remark:

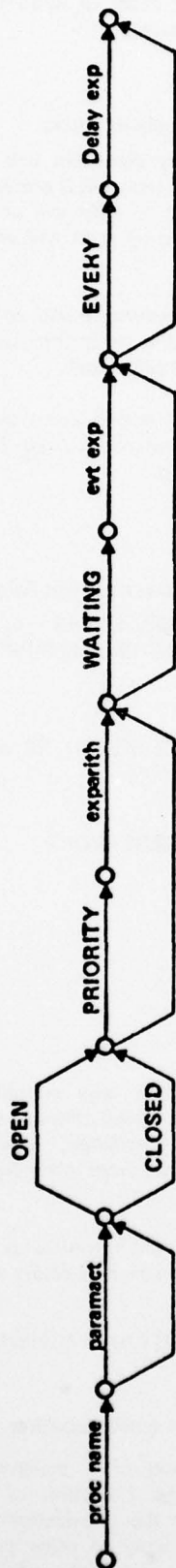
The process block may contain call statements to another (or the same) process so there may be calls which, upon compilation, correspond to an article which has not yet been compiled, which would cause errors.

A solution to this problem is found by making a DUMMY article which contains only name and formal parameters:

```
ARTICLE DUMMY name [formal parameters].
```

3.5.3. Process synchronization

During execution of a multitasking system, it may happen that the execution of certain processes are conditioned by the termination of other processes or by a wait for reply of other processes. Thus, it may be necessary to schedule, in time, the execution of certain programs, hence the necessity of synchronization primitives. In the MAXIRIS language, this has been done by means of EVENT and RESOURCE.



Proc name : Name of the process to be called
 Paramact : actual parameter (see graph)
 Exparith : Value of the priority (SHORT INTEGER type)
 Evt exp : Event expression (see graph)
 Delay exp : Delay expression (see graph)

PROCESS CALL

3.5.3.1. SYNCHRONIZATION USING THE EVENTS

An event is a signal sent out by the process undergoing execution. This signal may be received by several tasks waiting for it: these tasks will then be activated according to their priority.

A task can **ACTIVATE** or **WAIT** for an event or an event expression.

The events may also be used in the process call sequence by means of the **WAITING** operator; a new task is thus generated but is put into waiting for

events before being able to be executed.

3.5.3.2. SYNCHRONIZATION USING THE RESOURCES

A task may release (**FREE**) or reserve (**RESERVE**) a resource. A resource may be shared among n users.

A resource has 2 states: free and busy. When it is busy and another user wishes to reserve it, the latter is put into a waiting queue until one of the others users releases the resource.

4. LINKS WITH THE OUTSIDE

4.1. INPUTS-OUTPUTS

4.1.1. General remarks concerning the processing of inputs-outputs

Basic MAXIRIS allows the direct processing of exchanges with the outside at two levels:

- At the logical file level by means of **READ**, **WRITE** and their associated **FORMAT**. This allows making data exchanges with conversion and pagination, which is particularly useful in debugging phase.
- At the elementary level by means of:
 - peripheral description allowing, in the **ARTICLE SYSTEM DATA**, associating with symbolic names, numerical values corresponding to addresses or control word on the various buses of the machine so as to make the program independent of these specific numerical values.
 - direct input-output elementary orders: **INPUT** and **OUTPUT** making it possible to carry out a word transfer (*data or command*) or to initialize a transfer.

Between these two levels, the needs depend:

- on the computer (*Input-Output method and structure*),
- on the input-output supervisor,
- on the peripherals to be used,
- on the standard handlers of these peripherals.

4.1.1.1. PERIPHERALS SUPPORTED BY THE INPUT-OUTPUT SUPERVISOR

For these peripherals, it is sufficient to be able to execute the corresponding monitor call, associating it with a control block (CB) describing the modalities of the transfer.

As far as the control block (CB) is concerned, it could be declared either as:

- a structure if it is static,
- a set, if it is dynamic.

It must comply with the standard CB format for the supervisor used.

4.1.1.2. SPECIAL PERIPHERALS

They may be implemented from a special procedure (*or INTRPT PROCEDURE for a peripheral with interrupt-synchronization*) using the elementary input-output commands.

4.1.1.3. ANALOG ACQUISITION PERIPHERALS

They may be handled in a particular way as in 4.1.1.2. However, independently of the particularities of an analog chain, any acquisition operation may be considered as the succession of elementary actions:

- choice of the gain and the conversion mode,
- selection of the channel,
- measurement validity test,
- scale conversion,
- comparison with thresholds,
- storage in the data areas.

We are then led to define a logical description of the analog input-output according to a **FORMAT** associated with the **READ-WRITE** and able to be included in the **BASIC MAXIRIS** in the same way as the standard files, the elementary actions being described in a particular **FORMAT**. The **READ-WRITE** phases produce a call to a procedure which interprets the **FORMAT** field descriptors.

This procedure written in MAXIRIS or in assembler will be specific of the acquisition equipment used.

4.1.2. Inputs-Outputs with format

4.1.2.1. DECLARATION OF THE FORMAT

It has the format:

FORMAT name (D1, -----, Di)

Name: name of the format mentioned in a **READ-WRITE** statement,
Di: field descriptor corresponding to an elementary section.

4.1.2.2. INPUT-OUTPUT STATEMENT

READ } name 1, name 2
 WRITE } [variable name 1 ----, variable name i]

Name 1 } name of the logic file if described by
 } DEVICE IOUNIT in ARTICLE SYSTEM
 } DATA
 } processing procedure name if not
 } declared in ARTICLE SYSTEM DATA

Name 2 name of the format

Name } name of a single variable
 variable } name of an array

4.1.2.3. DATA INPUT-OUTPUT ON STANDARD FILES

The field descriptors:

— Describe the type of conversion which may be:

- Alphanumerical,
- Boolean,
- Floating-point,
- Integer,
- Logical,
- Quality,
- Reference,
- Spacing,
- Fixed point,

— Specify the length of the field.

— Specify the number of digits in the fractional part (*optional*).

Example:

```

FORMAT F1 (A18, F6.3, S3, Q4)
=====
WRITE IMP F1 [STRING1, ALPHA, COLOR]
  
```

4.1.3. Input-Output under supervisor control

4.1.3.1. EXECUTION

The request for an input-output execution is made by the statement:

IOCS [CB name, log name]

- CB name designates a structure, an array, or is a reference,
 - log name is an optional status word.
- Depending upon the sophistication of the supervisor, the input-output request is executed immediately (*and possibly refused*) or put into queue.

4.1.3.2. WAIT FOR END OF INPUT-OUTPUT

The corresponding statement has the form:

WAITIO [CB name, log name]

The arguments have the same significance as those of IOCS. This statement allows suspending the execution of the task until the input-output designated by the CB is terminated.

4.1.3.3. CONTROL BLOCK

It describes the input-output requested. Its format

depends on the input-output supervisor.

Example:

```

STRUCTURE CB1 (SHORT LOGICAL COMD :
HEX'000B';
NUMLOG : HEX'00FF'; REFERENCE BUFFER
ADBUF;
SHORT INTEGER LGBUF : 80); -----
-----
STRING 80 BUFFER ; -----
-----
LOCS CB1 ; WAITIO CB1 -----
  
```

4.1.4. Direct inputs-outputs

They may use the standard declarations and statements of the BASIC MAXIRIS.

4.1.4.1. DECLARATION

The declaration is made in ARTICLE SYSTEM DATA by: DEVICE DIRECT name, constant.

Name : symbolic name of the digital input-output.

Constant : will replace name in the input-output statements.

4.1.4.2. INPUT-OUTPUT STATEMENTS

INPUT
 OUTPUT [name, info, log name]

INPUT : direct input

OUTPUT : direct output

Name : symbolic name declared in ARTICLE SYSTEM DATA.

Info : symbolic name of a SHORT INTEGER or SHORT LOGICAL which will contain the information exchanged.

Log name : SHORT LOGICAL containing the status (*optional*).

Example:

```

DEVICE DIRECT CDLEC HEX 'OAF2';
DEVICE DIRECT INFLEC HEX 'OAF3';
  
```

```

SHORT LOGICAL MARCHE : HEX'000A', LECTUR;
OUTPUT [CDLEC, MARCHE] ; INPUT [INFLEC,
LECTUR] ;
  
```

4.2. INTERRUPTS

4.2.1. Declaration of an interrupt level

The declaration of an interrupt level is made as follows:

DEVICE INTRPT name No. of interrupt level;

Example:

```

DEVICE INTRPT INT 24;
  
```

4.2.2. Interrupt procedure

This is declared as follows:

- **ARTICLE INTERRUPT PROCEDURE** procedure name
BEGIN
END

It is a procedure which will be initiated by the occurrence of the interrupt associated with it.

4.2.3. Interrupt control statements

They allow controlling an external interrupt.

- **ATTACH** : associates an interrupt level to a procedure.
- **DETACH** : suppresses the previous association.
- **ITCTRL** : forces the status of the flip-flops according to a quality attribute.

The significance of the quality attribute depends on the hardware of the computer's interrupt system.

APPENDIX

GLOSSARY

Activation (*of an event*)

Language statement by which an event is made to occur.

Article

Constituent element of a program and of a library. Distinction is made between *data*, *procedure* and *process* articles.

Assembler

Processor for transforming a source program module into an object program module in relocatable binary format.

Association (*of a procedure and an interrupt*)

Action by which it is declared that a procedure is the one to be executed when a certain interrupt occurs.

Block

List of declarations and statements grouped between **BEGIN** and **END** to form one statement only. The declarations within a block have no reach outside of this block.

Closed call

Process call for which the calling task loses control and is authorized to take it back only when the called task is terminated. A closed call may be found inside a process only.

Creation

Action consisting in generation and initialization of a new element in a set.

Declaration (*of data*)

Definition of a data made by associating a type and

a format to a name.

Dynamic data

Data having a birth, a life and a death. They are the elements of the sets.

Eligible

Status of a task when it is likely to be chosen by the monitor's scheduling algorithm for execution.

Event

Basic entity of the language. An event is either occurred or non-occurred.

Field

Constituent object of a set or structure. The elements of a set are characterized by their fields.

Function

Value calculation sub-routine. A function replaces an expression when it is used. A function constitutes a procedure article.

Inputs-outputs

The control and checking operations for peripheral equipments and transfers.

Monitor

Set of procedures in charge of managing the tasks, the life data, the resources, the events and the input-outputs.

Open call

Process call for which the process, the procedure or the calling function does not lose control.

Procedure

Statement identified by a name and specifying an action. A procedure may be followed by formal parameters updated at the time of the call. A procedure constitutes an article.

Process

Same definition as a procedure. The difference lies in the call level. See closed call and open call.

Reference

A reference data contains the address of a set element.

Release

Action consisting, for a process or a procedure, in releasing the control of a resource. Any task that has been stopped by an unsuccessful attempt to reserve the resource becomes eligible again.

Reservation

Action by which a process attempts (*with success or not*) to take control of a resource.

Resource

Entity which may have two states: free and busy. A resource may be occupied by several users at the same time, it depends upon its parallelism (*i.e. the maximum number of users it has*).

Scheduling algorithm

Monitor procedure which, when in control, deter-

mines the task to be executed out of all the eligible tasks.

Set

Possible format of a data which is actually dynamic data. The elements of a set are created and deleted.

Statement

Language rule by which an action is specified.

Structure

Association of several elements forming a data block. Each element may be composed of several fields.

Task

Formed by the name of a process, the list of input parameter values. Whereas the process is essentially static, the task is dynamic and corresponds to the "job" of a process.

Type

Indicates the nature of a data.

Waiting

Status of a task waiting either for the arrival of an event or for the end of the execution of a task it created by a closed call, or the release of a resource.

ANNEX J

PEARL SUBSET FOR AVIONIC APPLICATIONS

CONTENTS

Para.	Title	Page
	INTRODUCTION	J-5
	PART I: SUMMARY OF THE MAIN CHARACTERISTICS OF PEARL	J-6
1.	SUMMARY OF THE LANGUAGE FEATURES	J-6
1.1	Features Offered by PEARL	J-6
1.1.1	Algorithmic Programming	J-6
1.1.2	Machine Independence	J-6
1.1.3	Parallel Programming (Tasking)	J-6
1.1.4	Modular Programming	J-6
2.	MAIN CHARACTERISTICS OF PEARL	J-6
2.1	Algorithmic Language Features	J-6
2.1.1	Data Representation	J-6
2.1.2	Data Definition and Assignment of Values	J-7
2.1.3	Operators and Expressions	J-8
2.1.4	Statements	J-8
2.1.5	Block Structure	J-9
2.2	Description of the Configuration and Input/Output	J-10
2.2.1	Structure of a PEARL Module	J-10
2.2.2	System Division	J-10
2.2.3	Input/Output Statements	J-12
2.3	Parallel Programming	J-13
2.3.1	Tasks	J-13
2.3.2	Declaration of Tasks	J-15
2.3.3	Control of the States of Tasks	J-15
2.3.4	Scheduling of Tasks Depending on Time Conditions and Asynchronous Events	J-15
2.3.5	Synchronization of Tasks	J-16
	PART II: ALGORITHMIC LANGUAGE FEATURES	J-17
3.	LANGUAGE ELEMENTS	J-17
3.1	Character Set	J-17
3.2	Literals	J-17
3.3	Computational Constants	J-17
3.3.1	Integers	J-17
3.3.2	Real Numbers	J-17
3.3.3	Bit Strings	J-18
3.3.4	Character Strings	J-18
3.3.5	Clock Constants	J-19
3.3.6	Duration Constants	J-19
3.4	Identifiers	J-19
3.5	Comments	J-20
4.	DEFINITION OF PROGRAM ENTITIES	J-20
4.1	Valid Scopes of Definitions	J-20
4.2	Variables	J-21
4.3	Attributes	J-21
4.3.1	Arrays	J-21
4.3.2	Mode Attributes	J-22
4.3.2.1	Modes of Non-Composite Data	J-22
4.3.2.2	Structures	J-22
4.3.3	INIT Attribute	J-23
4.3.4	INV Attribute	J-24
4.3.5	Relations between Modes and Entities	J-24
4.3.6	IDENT Attribute	J-25
4.3.7	GLOBAL Attribute	J-25
4.3.8	RESIDENT Attribute	J-26
4.4	Definitions at Module Level	J-26
4.4.1	Declaration of Standard Lengths	J-26
4.4.2	Specifications	J-26
4.4.3	Declaration of Data at Module Level	J-27
4.5	Declarations at Task, Procedure and BEGIN-Block Level	J-28

Para.	Title	Page
5.	OPERATORS AND EXPRESSIONS	J-29
5.1	Operators	J-29
5.1.1	Monadic Operators	J-29
5.1.2	Dyadic Operators, Order of Precedence	J-29
5.2	Expressions	J-30
5.2.1	Arithmetic Operations	J-31
5.2.2	Logical Operations	J-31
5.2.3	Comparison Operations	J-32
5.2.4	Concatenation Operations	J-32
5.2.5	Shift Operations	J-33
5.2.6	Conversion Operations	J-33
6.	STATEMENTS	J-33
6.1	Assignment	J-34
6.2	BEGIN-Block	J-36
6.3	Sequential-Control	J-36
6.3.1	SKIP-Statement	J-37
6.3.2	GOTO-Statement	J-37
6.3.3	Conditional-Statement	J-37
6.3.4	CASE-Statement	J-38
6.3.5	REPEAT-Statement	J-38
6.3.6	CALL-Statement	J-40
6.3.7	RETURN-Statement	J-40
7.	PROCEDURES	J-41
7.1	Procedure Attributes	J-41
7.2	Procedure Declaration	J-41
7.3	Procedure Specification	J-43
7.4	Procedure Call	J-43
7.5	Standard Functions	J-46
	PART III: INPUT/OUTPUT FACILITIES	J-47
8.	THE SYSTEM DIVISION	J-47
8.1	Configuration Description in the System Division	J-47
8.2	Connection of Single Devices	J-49
8.3	Connection of Device-Arrays/Groups	J-49
9.	DEVICES	J-51
9.1	Device Attributes	J-51
9.2	Declaration of Devices	J-51
9.3	Specification of Devices in the Problem Division	J-52
10.	FILES (DATA STATIONS)	J-52
10.1	Definition of Files (Distinction between Devices and Files)	J-52
10.2	File Attributes	J-52
10.3	Declaration and Specification of Files	J-54
10.4	Creation and Deletion of Files	J-54
10.5	Opening and Closing of Files	J-55
11.	INPUT/OUTPUT OF PROCESS SIGNALS	J-56
11.1	Statements for Direct Process Input/Output	J-56
11.2	Statements for Organized Process Input/Output	J-56
12.	STANDARD INPUT/OUTPUT	J-57
12.1	Composition of Formatted I/O Statements	J-57
12.2	Formats	J-57
13.	GRAPHIC INPUT/OUTPUT	J-59
13.1	Composition of Graphic I/O Statements	J-59
13.2	Graphic Formats	J-60

Para.	Title	Page
	PART IV: PARALLEL PROGRAMMING FEATURES	J-61
14.	TASKS	J-61
14.1	Introduction of Tasks	J-61
14.2	Declaration of Tasks	J-61
14.3	Specification of Tasks	J-61
15.	SCHEDULING OF TASKS	J-62
15.1	Time-Dependent Scheduling	J-62
15.2	Event-Dependent Scheduling	J-63
16.	TASK CONTROL STATEMENTS	J-63
16.1	Activation of Tasks	J-64
16.2	Suspension of Tasks	J-64
16.3	Continuation of Tasks	J-65
16.4	Resumption of Tasks	J-65
16.5	Termination of Tasks	J-66
16.6	Prevention of SCHEDULE Conditions	J-66
17.	SYNCHRONIZATION OF TASKS	J-66
17.1	Semaphore Variables	J-66
17.2	Declaration and Specification of Semaphores	J-67
17.3	Semaphore Operations	J-67
	APPENDIX 1	J-68
	APPENDIX 2	J-71
	REFERENCES	J-96

AD-A044 915

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/G 3/1
A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CON--ETC(U)
MAY 77 G E SCHWEIZER, A A CALLAWAY, E C GANGL

UNCLASSIFIED

AGARD-AR-90

NL

5 OF 6
ADA
044 915



PEARL SUBSET FOR AVIONIC APPLICATIONS

INTRODUCTION

Embedded computer systems with realtime constraints play a major role in today's avionic applications. Both users and implementers of avionic systems are plagued with increasing problems within the software area and the interface between system architecture and software design. The same is true for other application areas of process automation with realtime computers.

The realtime language PEARL offers an attempt to alleviate and to tackle these problems, which normally emerge in form of high cost, delayed development schedules, incompatibilities and difficulties with retrofit programs. The goal of PEARL is to provide efficient tools for the systems engineer, which enable him to write, test and maintain effective process computer programs by himself, without the help of a specialized software team. For this purpose PEARL has been designed to be easy to learn, very reliable and self-documenting.

PEARL is a procedure-oriented programming language, which is not limited to specific application areas. Automation tasks for all kinds of technical processes may be programmed in PEARL, taking advantages of the fact that PEARL provides clear interfaces between hardware and software.

Automation programs written in PEARL may be easily transported to different process computers. The portability of PEARL programs, even on process computer systems with different process peripherals, is obtained by means of language features (the so-called System Division). Thus PEARL programs are portable and reusable when the type of process computer is changed.

The portability of PEARL automation programs with respect to realtime properties is guaranteed by the fact that the realtime features are included at the language level. Specific properties of the realtime operating systems are predefined through the language elements of PEARL and a standardization of realtime operating systems properties is achieved.

The application of PEARL will substantially increase cost-effectiveness of software development and maintenance and in particular will:

- reduce coding, debugging and checking time and cost,
- make software production more visible and manageable, and
- provide the desired flexibility and freedom for system updates and retrofits.

This document, describing the PEARL-Subset for Avionic Applications, comprises:

PART I	SUMMARY OF THE MAIN CHARACTERISTICS OF PEARL
PART II	ALGORITHMIC LANGUAGE FEATURES
PART III	INPUT/OUTPUT FACILITIES
PART IV	PARALLEL PROGRAMMING FEATURES
APPENDIX 1	Description of the meta-language used in the formal syntax notation
APPENDIX 2	PEARL features of the Subset for Avionic Applications in formal syntax notation.

PART I: SUMMARY OF THE MAIN CHARACTERISTICS OF PEARL

1. SUMMARY OF THE LANGUAGE FEATURES

1.1 Features Offered by PEARL

The main features offered by the real-time programming language PEARL can be explained under the following headings:

1.1.1 Algorithmic Programming

The programming of algorithms is not a typical attribute of process control. In this respect there are no significant differences between the requirements of data processing for the control of industrial processes and the requirements of data processing for scientific, technical or commercial applications. Therefore, the algorithmic programming features of PEARL have been taken mainly from other languages well proven by implementation and application (e.g. from PL/I).

1.1.2 Machine Independence

In order to be able to formulate automation tasks as independently as possible from any manufacturer-specific process interface system, a PEARL program is divided into two parts, the system division and the problem division.

The system division provides the description of the configuration of the process peripherals, and the declaration of symbolic names for the process signals and peripheral elements. These symbolic names are then used in the problem division (the automation program itself). The symbolic names are user defined and should be chosen to be meaningful. If a PEARL program is transferred to another type of process control computer with another process interface system, this scheme allows the problem division to remain as it is and only the system division need be adapted.

1.1.3 Parallel Programming (Tasking)

The method of parallel programming, also known as asynchronic programming, is used to fulfil the real-time requirements resulting from the use of process control computers⁴. In order that parallel programming can be performed, PEARL offers statements for

- the declaration of tasks (task name, priority etc.),
- the control of the transitions between task states,
- the singular or repetitive scheduling of tasks depending upon time conditions or the occurrence of interrupt signals,
- the synchronization of tasks.

1.1.4 Modular Programming

A PEARL program system can be constructed from a set of separately compilable units. These are called PEARL program modules. Such a module normally contains a system division and a problem division, but can also consist of just a system division or just a problem division (e.g. a procedure).

The symbolic names for process signals or process terminals, as introduced in the system division of a module, must be declared as being global in all modules of a PEARL program system. The connections between the problem divisions of different modules are thus realized.

2. MAIN CHARACTERISTICS OF PEARL

2.1 Algorithmic Language Features

2.1.1 Data Representation

Character Set

The character set consists of the 26 alphabetic (A-Z), 10 numeric (0-9) and some additional special characters.

Names (identifier)

Names must consist solely of alphanumeric characters and the first character of a name must be alphabetic.

Comments

Comments may be included in the program text. The comment is preceded by `"/*` and followed by `*/`, i.e. `/*comment*/`.

Data Types

FIXED	integer number
FLOAT	floating point number
BIT(n)	bit string with n bits
CHAR(n)	character string with n characters
CLOCK	time (of day)
DUR	interval of time

Representation of Constants

The following table demonstrates the representation of constants for the above mentioned data types.

Representation of Constants in PEARL

<i>Data type</i>	<i>Constant consists of</i>	<i>Example</i>
FIXED	one or more decimal digits	233
FLOAT	floating point number in conventional or exponential notation	0.233 23.3E-2
BIT	one or more binary digits, enclosed in quotation marks and followed by "B1"; one or more octal digits, enclosed in quotation marks and followed by "B3"	'11101001' B1 '351' B3
CHAR	one or more characters from the full character set, enclosed in quotation marks ('is represented by')	'TEST_1'
CLOCK	time of day, in hours, minutes and seconds, delimited by colons (:))	17
DUR	duration in hours, minutes and seconds, delimited by HRS, MIN and SEC respectively	2 HRS 25 MIN 31 SEC

2.1.2 Data Definition and Assignment of Values

Declarations and Specifications

Variable data, to be used in a program, must be defined by a data name and a data type. Constant data, which cannot be changed during the program run must be declared as such by using the keyword INV (denoting invariable).

There are two types of data definitions:

The first type, "Declaration", is used in order to introduce a variable for the first time. For this purpose the keyword DECLARE (abbreviated: DCL) is used.

If a variable is to be common to more than one module, it must be specified as GLOBAL. Initial values can be given to variables using the keyword INITIAL.

Example: DECLARE (A, B, C, D) FLOAT INITIAL (0.1, 0.2, 0.3, 0.4);
means that the floating point variables A, B, C, D are declared with the initial values
A = 0.1, B = 0.2, C = 0.3, D = 0.4.
DCL PI INV FLOAT INIT (3.14159);
means that the value 3.14159, in floating point format, will be assigned to the variable
named PI.

The second type, "Specification", is used in order to define a variable, the declaration of which is expected to be made in another module. To do this the keyword SPECIFY (abbreviated: SPC) is used.

Example: SPECIFY I FIXED GLOBAL;
means that the integer variable I is specified, and must already have been declared in
another module as a global variable.

Assignment of Values

Assignment statements are used to assign values of a certain type to variables during computation. The variable to the left of the assignment symbol, "=", must be of the same data type as the constant, variable or expression to the right (homogeneity rule).

Examples:

Y = 10E*7E-3;	Type: FLOAT
I = J/K	FIXED OR FLOAT
A = '101'B1 AND '3'B3;	BIT
TEXT = 'MESSWERT';	CHAR

2.1.3 Operators and Expressions

Operators

PEARL offers a variety of operators that simplify data handling.

Arithmetic Operators	Comparison Operators	Logical Operators
** Exponentiation	LE less than or equal to	AND logical AND
* Multiplication	GE greater than or equal to	OR logical OR
/ Division	LT less than	NOT logical NOT
+ Addition	GT greater than	(negation)
- Subtraction	EQ equal	

Conversion Operators	Bit/String handling Operators
FIT	CAT Concatenation
CHAR	SHIFT logical shift
FIX	CSHIFT cyclic shift
FLOAT	
BIT	

Expressions

An expression is composed of one or more operands (variables, constants) connected to each other by operators. Normally, all operands within an expression have to be of the same type (rule of homogeneity).

2.1.4 Statements

Branch Statements and Labels

By using the unconditional branch statement, GOTO, the linear flow of the program is changed and the program continued at another position, the branch target. Branch targets are marked by labels which are denoted in the same way as the names of variables.

```

Example:      GOTO M10;
               .
               .
               .
M10   :   A=5.1;
  
```

Conditional Statements

By using conditional statements the program flow may be altered, depending upon the value of an expression of type BIT(1):

```

General format: IF binary expression of type BIT(1)
                THEN statement(s);
                ELSE statement(s);
                FIN;
  
```


Examples:

```

DCL MODESWITCH BIT(1);      DCL(A,B) FLOAT;
.
.
.
.
IF MODESWITCH
THEN GOTO MANMOD;
ELSE GOTO AUTMOD;
FIN;

IF A LE B
THEN GOTO LABX;
FIN;

```

Loop Control Statements

Loop statements can be used to simplify the coding of repetitive procedures.

General format:

```

FOR          loop control variable
FROM         initial value
BY           step
TO           final value
REPEAT       statement(s); (to be repeated)
END;

```

Example:

```

FOR I FROM 0 TO 10 BY 1 REPEAT
J = J + 4*I**2;
END;

```

The effect of this loop is the repetitive computation of $J = J + 4I^2$ for the values $I = 0, 1, \dots, 10$.

2.1.5 Block Structure

PEARL, in the same way as ALGOL and PL/I, offers the facility to subdivide programs into "blocks". "Blocks" are enclosed within the key-words BEGIN and END.

For the declaration of variables the same rules apply as in ALGOL or PL/I:

A variable, which is declared within a block, is only known within that block or within any inner blocks, but not in any outer blocks.

The formulation of procedures and function procedures is also possible.

Function procedures produce single value results with or without input parameters.

General format:

```

Procedure name: PROCEDURE (List of input parameters with their resp. data types)
                  RETURNS (Data type of returned value);
                  Declaration of internal entities;
                  Statements;
                  RETURN (Expression)*;
                  END;

```

Example: Computation of the factorial of an integer by means of the function procedure FAKULT.

Declaration of the procedure:

```

FAKULT: PROCEDURE (N FIXED) RETURNS (FIXED);
DCL (I,NF) FIXED;
NF = 1;
IF N = 0 THEN RETURN (NF);
ELSE BEGIN
  FOR I FROM 1 TO N BY 1 REPEAT
    NF = NF * I;
  END;
FIN;
RETURN (NF);
END;

```

* RETURN statements are placed at the logical end of the function procedure. When executing any return the value of the function will be defined as the value of the expression in the brackets. This value must be of the same data type as that specified after the key-word RETURNS in the procedure declaration.

Use of FAKULT within a program:

K = FAKULT (M);

Effect: Computation of M! and assignment to the variable K (M has to be declared as FIXED in the calling program).

Procedures cause the execution of a specified sequence of statements, possibly dependent upon a set of input arguments.

General format:

Procedure name: PROCEDURE (List of arguments with their resp. data types);
Declarations;
Statements;
END;

Example: Combination of a frequently used sequence of statements into the procedure UP.

UP: PROCEDURE (L BIT(1), A FIXED, B FIXED);
IF L THEN A = A + B;
ELSE A = A * B; FIN;
SEND FROM A TO DIGO2;
END;

The procedure is used with a CALL statement, e.g. by CALL UP (F,C,D).

The actual parameters F,C,D must have been declared in the calling program with the types BIT(1) for F and FIXED for C and D.

2.2 Description of the Configuration and Input/Output

2.2.1 Structure of a PEARL Module

A PEARL program can consist of a series of separately compilable units, i.e. PEARL modules. Together with the operating system the compiled modules represent an executable PEARL program system.

The structure of a PEARL module is shown in Figure 2.1.

The system division describes the configuration of the devices of the process computer system and its connections to the process being controlled. The problem division contains the formulation of the automation problem itself and can be written independently using the names defined in the system division.

These names must have been specified prior to their use in the problem division.

2.2.2 System Division

The connections between the devices of the process computer system, the assignment of the interface connections to the process signals, and the direction of the information flow are specified in the system division. The different terminals and process signals take various symbolic names which are then used in the problem division.

Device connections are described as follows:

device description	connection direction symbol	device description
-----------------------	--------------------------------	-----------------------

The following symbols are used to indicate the direction of data flow:

A → B	Transfer from A to B
A ← B	Transfer from B to A
A ↔ B	Transfer in both directions

For the device description there is a set of fixed key-words. A key-word is associated with each type of device used in the process computer system. In addition to the device description a connection number is also allocated whenever several other devices are connected to that device.

Example: CENTPROC * 4 → CONTRUNIT;

indicates that I/O channel 4 of the central processing unit, called CENTPROC, is connected with the device CONTRUNIT.

If a symbolic name has to be assigned to a device this name must be specified immediately before the relevant device type.

Example: CENTPROC * 7 → TYPEWRITER: ASR33;

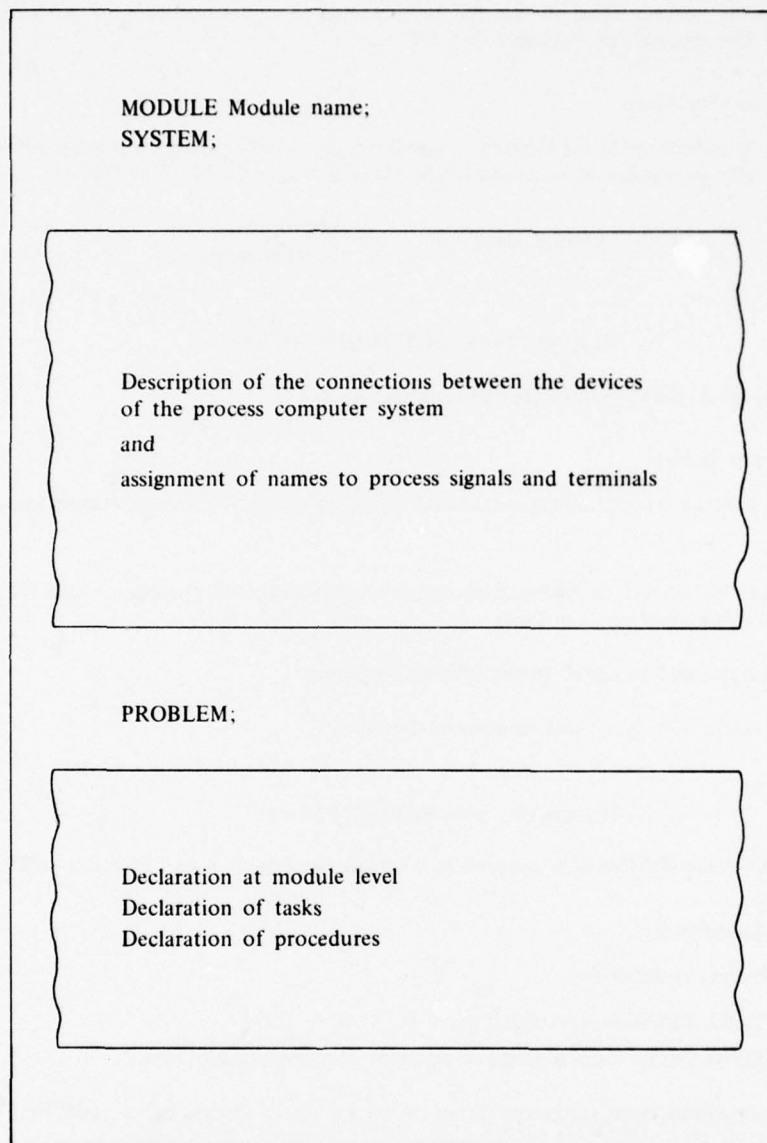


Fig.2.1 Structure of a PEARL Module

Assignment of Interface Connections to Process Signals

The assignment of a process signal to a device is achieved as follows:

device identifier: $\left\{ \begin{array}{c} \longrightarrow \\ \longleftarrow \\ \longleftrightarrow \end{array} \right\}$ device description

For the identification of process signals and process terminals meaningful symbolic names may be introduced.

Additional device information is required if more than one process signal is connected to a device. The additional information required is as follows:

- the number of the connection,
- for digital input/output, the number of the first bit and the number of bits belonging to the process signal.

Example:

WARNING: ← DIGOUT * 3 * 5,1;

means that the process signal WARNING is connected to connection 3, bit 5 of the digital output device DIGOUT. The process signal consists of 1 bit.

Assignment of Names to Interrupts

Interrupts serve to indicate external events (i.e. spontaneous events occurring normally outside the process control computers). The assignment of names to interrupts is normally described as follows:

interrupt identifier: → interrupt receiving
device descriptor;

Example:

ALARM: → INTERRUPT (3);

The interrupt signal ALARM is connected to interrupt No.3.

Assignment of Names to Signals

Signals serve to indicate internal events (i.e. events occurring within the process control computers).

Example:

The message "overflow", occurring if the result of an arithmetical operation exceeds the boundaries of the representable numbers, is a signal.

Identifiers can be assigned to signals in the following manner:

signal identifier: → signal receiving device descriptor;

Example:

IOERROR: → ERRORTYPE (4);

A signal with the name IOERROR is assigned to a certain error type in an I/O device (ERRORTYPE(4)).

2.2.3 Input/Output Statements

I/O Statements for Process Peripherals

Input: TAKE FROM process signal name TO variable name;

Output: SEND FROM variable name or constant TO process signal name;

Example of an analogous input statement (input of the analogous process signal TEMPERATUREFEELER):

TAKE FROM TEMPERATUREFEELER TO RAWDATA;

Example of a digital output statement (output of a binary value to the process signal WARNING):

SEND FROM A TO WARNING;

(A must have been defined to be of type BIT(1).)

I/O Statements for Standard Peripherals

Input: GET FROM device identifier TO data list THROUGH format list;

Output: PUT FROM data list TO device identifier THROUGH format list;

Each element of the data list must have a corresponding element in the format list. The format list contains details of how the input information is to be interpreted and in what form the internal bit string is to be represented.

The following table shows the various format elements and examples of their use. More examples of the use of I/O statements for the standard peripherals are contained in the program example in Chapter 12.

Format Elements and Examples

Type of data elements	Format element	Significance of the format element	Example	
			Data list	format
FIXED	F(Y)	Input/Output of a fixed-point number (field width of Y chars. including sign).	-30	F(3)
FLOAT	E(Y,Z)	Input/Output of a floating-point number (field width Y chars.) in exponential notation. Z chars. of the mantissa are right of the decimal point.	-4.27E10	E(8,2)
	F(Y,Z)	Input/Output of a floating-point number (field width of Y chars. including sign and decimal point).	427.35	F(6,2)
BIT	B1(Y)	Input/Output of a bit string as a binary number with Y bit positions.	1001001	B1(7)
BIT	B3(Y)	Input/Output of a bit string as an octal number with Y character positions.	765	B3(3)
CHAR	A(Y)	Input/Output of a character string with Y character positions.	NEW VALUE	A(9)
CLOCK	T(Y)	Input/Output of a time	10:40:47	T(8)
DUR	D(Y)	Input/Output of a time period	20 HRS 07 MIN 21 SEC	D(17)
control elements	X(Y)	Output of Y space characters		
	SKIP	Output of new line		
	PAGE	Output of new page.		

2.3 Parallel Programming

2.3.1 Tasks

Definition of a Task

Program systems for the automation of technical processes have to fulfil the following requirements:

- The programs for the realization of the automation task have to run at the time required by the process. This is true for the automation functions which are tied to fixed times or time intervals (e.g. supervision of measured values in certain time intervals) as well as for those which must be performed at unforeseeable times (e.g. evaluation of alarms) (Ref.6).
- For the automation of concurrent partial processes the corresponding automation functions must also be treated concurrently. At least it has to be ensured that program runs which must be parallel, are executed "as if" they were parallel (quasi-parallel).

For parts of a program to be executed in a "quasi-parallel" manner in PEARL the notion of a "task" was introduced.

Definition:

A task is the realization of an automation task by a program whose execution is controlled by the operating system.

States of Tasks

In order to describe the chronological order of tasks the notion of a state of a task should be introduced. A task can have the following states (see Figure 2.2):

- State "running":
The task is running, i.e. the processor is delegated to the task.

- State "runnable":
The conditions required for the running of the task are fulfilled (e.g. time conditions, events, etc.), but the processor has not been delegated to the task by the operating system.
- State "suspended":
The task is relegated for a certain time or until the occurrence of an event.
- State "dormant":
The task is known to the operating system, but no order to execute it has been given.

Synchronization of Tasks

By demanding the execution of the tasks at predetermined times, at certain time intervals, or on the occurrence of certain events, it should be ensured that the tasks are executed synchronically to the events in the technical process.

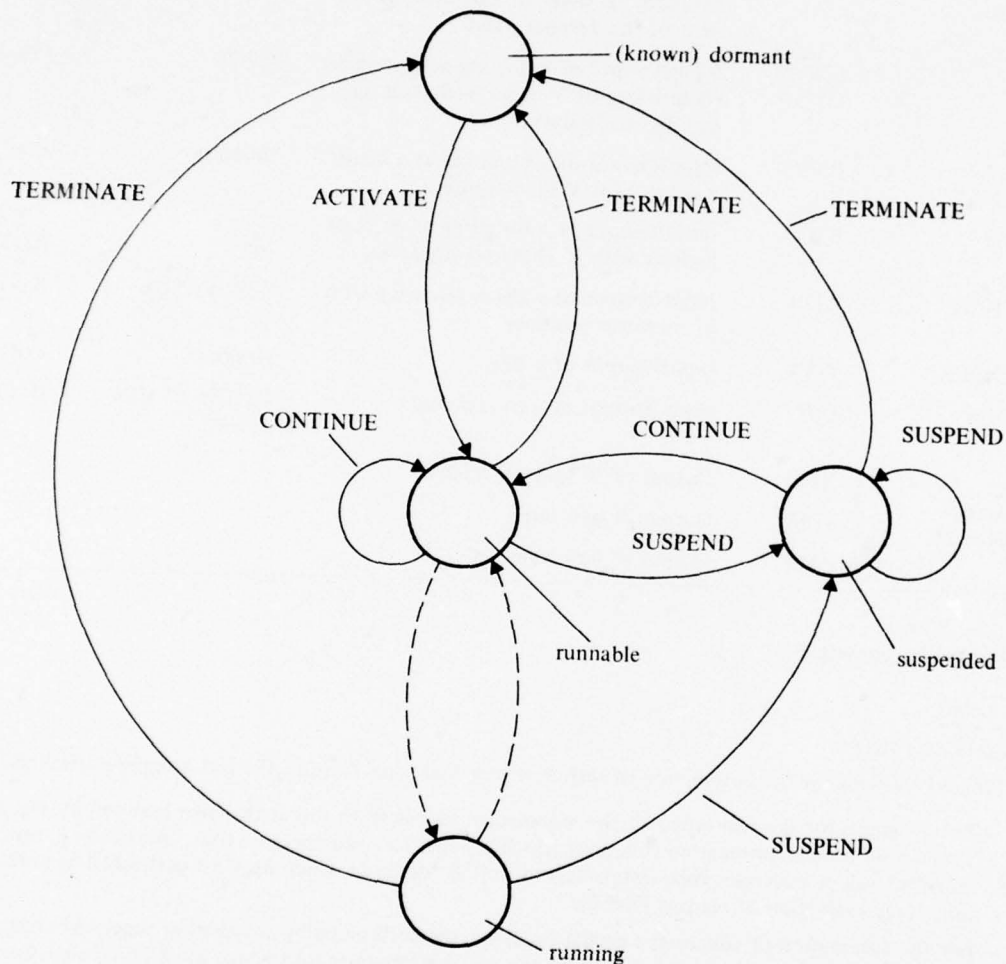


Fig.2.2 Transitions between States of Tasks in PEARL.

Solid lines: transitions activated through control statements.
Dotted lines: transitions activated through the system.

However, with the control of the quasi-parallel execution by the operating system, delays cannot always be avoided and the synchronism between the tasks and events in the process may be disturbed. This may not be desirable, particularly if the order of the execution of the tasks is important to the function of the technical process.

In these cases care has to be taken that the scheduling of the tasks by the operating system does not change the required logical order. The means by which disturbances of the synchronism between the technical process and the tasks can be avoided is called (logical) synchronization⁴. In PEARL, two kinds of synchronization variables are provided for the synchronization of tasks. These are:

- Semaphore variable
- Bolt variable.

The PEARL subset described herein only deals with semaphore variables.

2.3.2 Declaration of Tasks

In PEARL tasks are declared with a name and a priority, as follows:

```
task label: TASK [PRIORITY priority number] ;
            task-intern declarations;
            statements;
            END;
```

2.3.3 Control of the States of Tasks

The most important statements provided for the control of the states of tasks are as follows. For a complete and full description refer to Part IV.

Activation statement:

```
ACTIVATE task label [PRIORITY priority designation] ;
action:   The task is transferred to the state "runnable".
```

Suspension statement:

```
SUSPEND task label;
action:   The task is transferred from the state "runnable" to the state "suspended".
```

Continuation statement:

```
CONTINUE task label;
action:   Continuation of a suspended task, i.e. transfer from the state "suspended" to the state "runnable".
```

Delay statement:

```
AFTER:    time duration variable RESUME;
action:    The task is transferred to the state "suspended" and is re-transferred after the stated time delay
            to the state "runnable".
```

Terminate statement:

```
TERMINATE task label;
action:    A task is transferred from the state "running" or "runnable" to the state "dormant". At the
            logical end of a task there must be a terminate statement. If no task label is specified at a
            suspension or termination statement, the statement is referred to the task in which it is executed.
```

2.3.4 Scheduling of Tasks Depending on Time Conditions and Asynchronous Events

Constants and Variables for Scheduling

The clock time constant designates a fixed time.

structure:	hours:	minutes:	seconds
example:	1 :	36 :	25

The clock time variable designates a variable whose value corresponds to a clock time.

Declaration: DCL variable name DUR;

Example: DCL WAITINGTIME DUR;

Task Control Key-Words

Execution of tasks at specified times:

AT	clock time expression (time of the first call)
ALL	duration expression (cycle time)
UNTIL	clock time expression (time of the last call)

Execution of tasks at certain time intervals:

AFTER	duration expression (time until the first call)
ALL	duration expression (cycle time)
DURING	duration expression (time until the last call)

Key-Words for the Control of Tasks by Interrupt Signals

Single execution of a task:

WHEN	name of the interrupt signal
------	------------------------------

Multiple execution of a task:

WHEN	name of the interrupt signal
ALL	duration expression
UNTIL	clock time expression or
DURING	duration expression

The application of the key-words to the task control is explained, with examples, in the table at the end of Part I.

2.3.5 Synchronization of Tasks

To enable task synchronization semaphore and bolt variables are provided. In the following paragraphs only the semaphore variables are explained.

Semaphore Variable

Declaration: DCL variable name SEMA INITIAL (integer constant);

An initial value may be given to the semaphore variable at the time it is declared. The value of the variable can only be changed later using semaphore operations.

Semaphore Operations

REQUEST sema-variable;

The value of the sema-variable is decremented by 1 if the result is not negative. If the result would be negative, the operation is delayed and the calling task suspended until the value of the sema-variable becomes positive.

RELEASE sema-variable;

This operation increments the value of the sema-variable by 1.

Examples for the Application of Key-Words for Task Control

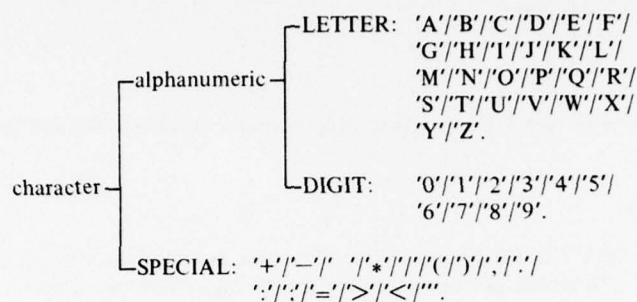
<i>Statement</i>	<i>Action</i>
ACTIVATE A;	Task A is transferred to the "runnable" state on execution of the statement (activation).
AFTER 5 SEC ALL 7 SEC DURING 106 SEC ACTIVATE B;	Task B is activated for the first time 5 seconds after the activation statement. The cyclic order ends 106 seconds after execution of the first activation.
AT 10:0:0 ALL 30 MIN ACTIVATE C PRIORITY 1;	Task C is activated from 10 o'clock on, every 30 minutes, with priority 1.
WHEN ALARM ACTIVATE D PRIORITY 3;	On the occurrence of the interrupt ALARM task D is to be activated with priority 3.
WHEN WARN ALL 1 MIN UNTIL 11: 0:0 ACTIVATE E PRIORITY 3;	On the occurrence of the interrupt WARN task E is to be activated until 11 o'clock, with a cycle time of 1 minute, and a priority of 3.
F:TASK;	
SUSPEND;	The statement suspends task F;
SUSPEND G;	This statement suspends task G;
END;	

PART II: ALGORITHMIC LANGUAGE FEATURES

3. LANGUAGE ELEMENTS

3.1 Character Set

Characters are the basic elements of the language. By combining characters other language elements can be built up which is analogous to the construction of words from letters in a natural language.



NB: The complete set of special characters is installation dependent.

3.2 Literals

Literals consist of one or more characters. Their character sequences are fixed (in contrast to data and identifiers) and have defined meanings. A list of all literals is found in Appendix 2.3. Literals which consist of alphanumeric characters are called key words. Key words are reserved, i.e. they cannot be used as identifiers. Other literals are composed of one or more special characters and act as operators or other syntactical items, e.g. punctuation marks.

Examples:

key-words	syntactical items
TASK	'
SIGNAL	;
B	=
	+
	**
	/

3.3 Computational Constants

Computational data can be loosely classified as arithmetic, string or time data.

3.3.1 Integers

Integer numbers are one form of arithmetic constants. Their mode (i.e. type) is 'FIXED'. They are written in decimal.

INTEGER-CONSTANT:
 DIGIT *.

Examples: 7845
0
1
008

+5 is an expression-seven (see 5.2) consisting of the operator "+" and the integer number "5" as operand.

3.3.2 Real Numbers

Real numbers are another form of arithmetic constants. They are of mode 'FLOAT'. They must contain either a decimal point or an exponent or both. The exponent consists of the letter E followed by a signed or unsigned integer constant.

REAL-CONSTANT:
 (DIGITS'.'DIGIT*/DIGIT*') 'E' ['+'/'-'] DIGIT*/
 DIGIT*'E' ['+'/'-'] DIGIT*.

Examples: 0.
 .0
 0.E12
 3.14159
 18.E-13
 .21E+3
 77E-04

+5.E0 is an expression-seven (see 5.2) consisting of the monadic operator "+" and the real number "5.E0" as operand.

3.3.3 Bit Strings

Bit strings are of mode 'BIT' ('INTEGER-CONSTANT') where integer constant gives the number of binary digits building up the string. Bit strings can be regarded as logical data. The logical values "true" and "false" are represented by a bit string of length 1.

BIT-STRING-CONSTANT:
 ""BINARY-DIGIT*"" ('B'/'B1')/
 ""OCTAL-DIGIT*"" 'B3'/
 ""SEDECIMAL-DIGIT*"" 'B4'.

Where:

BINARY-DIGIT: '0'/'1'.

OCTAL-DIGIT: '0'/'1'/'2'/'3'/'4'/'5'/'6'/'7'.

SEDECIMAL-DIGIT: DIGIT/'A'/'B'/'C'/'D'/'E'/'F'.

An octal digit represents
 three binary digits, a
 sedecimal digit represents
 four binary digits.

octal	binary	sedecimal	binary
0	000	0	0000
1	001	1	0001
2	010	2	0010
3	011	3	0011
4	100	4	0100
5	101	5	0101
6	110	6	0110
7	111	7	0111
		8	1000
		9	1001
		A	1010
		B	1011
		C	1100
		D	1101
		E	1110
		F	1111

Examples:

binary	octal	sedecimal	mode
'11001101011'B1	'7153'B3	'E6B'B4	BIT(12)
'110010100'B1	'624'B3	'194'B4	BIT(9)
'1101'B	'15'B3	'D'B4	BIT(4)
'110'B1	'6'B3	'6'B4	BIT(3)
'0'B	'0'B3	'0'B4	BIT(1)

3.3.4 Character Strings

Character strings are of mode 'CHAR' ('INTEGER-CONSTANT') where integer constant gives the number of characters.

CHARACTER-STRING-CONSTANT:

""CHARACTER \$"".

The characters which can occur in a string are letters, digits and special characters. An apostrophe within a character string is marked by another apostrophe.

Examples:

<i>string</i>	<i>mode</i>
'SCAN XY'	CHAR(7)
'1'	CHAR(1)
'OPERATOR' 'S'	CHAR(10)

3.3.5 Clock Constants

Clock constants are one form of time data and are of mode 'CLOCK'. They consist of three integer constants. The first gives the number of hours interpreted as modulo 24, the second the number of minutes, and the third the number of seconds. The numbers for minutes and seconds must not be greater than 59.

CLOCK-CONSTANT:
DIGIT*':'DIGIT*':'DIGIT*.

Examples: 0:0:0
 12:0:02
 25:59:3

3.3.6 Duration Constants

Duration constants represent time intervals and are of mode 'DUR'.

DURATION-CONSTANT:
DIGIT*'HRS'+DIGIT*'MIN'+DIGIT*'SEC'+DIGIT*'MSEC'.

Examples: 200 MIN 04 SEC
 1224 HRS 34 MIN 765 SEC
 190 HRS
 132 SEC
 1 HRS 0 MIN

3.4 Identifiers

Entities (devices, files, semaphores, interrupts, signals, variables, arrays, structures, libraries, formal parameters and labels) must be represented by names. Names are associated with entities in definitions and are called identifiers.

IDENTIFIER: LETTER (LETTER/DIGIT) \$.	see 3.1
---	------------

Only the first six characters of the identifier are significant. Identifiers may be chosen by the user, however, some character sequences cannot be used as they are reserved for key-words (see Appendix 2.3). See also "4.1 Valid Scopes of Definitions" and "7.5 Standard Functions".

Labels indicate entry points to statements, tasks, and procedures. The syntactical term is LABEL-IDENTIFIER.

If several entities with the same attributes (see 4.3) have to be defined their identifiers may be put together in a single bracketed list followed by their common attributes. This list is expressed by:

ONE-IDENTIFIER-OR-LIST:
IDENTIFIER/'('IDENTIFIER //','')'.

*Examples:*Correct use of identifiers:

- 1) SPEED
- 2) ANGLE1
- 3) DCL (PITCH, ROLL, YAW) FLOAT;
Declaration of three entities of mode FLOAT.

Incorrect identifiers:

- 4) ROLL-ANGLE Only letters and digits are permitted.
- 5) ANGLE 1 Blanks are not permitted.
- 6) FACTOR1 The first six characters of each identifier are the same, so that
FACTOR2 the program cannot distinguish between them.

3.5 Comments

Comments can be inserted into the program text between language elements. They must be delimited by /* and */. Therefore */ cannot be used in a comment. A comment can be continued over several lines, the terminating delimiter being required only at the end of the last line of the comment.

Example: /*THIS IS A COMMENT*/
 /*COMMENT MAY BE
 SEVERAL
 LINES LONG*/

4. DEFINITION OF PROGRAM ENTITIES

All identifiable entities must be defined in each module in which they are used. An entity may be defined in a module once only.

For certain entities it is necessary to distinguish between two kinds of definition, declaration and specification.

A declaration is a definition which actually assigns areas of store to the entities.

A specification just indicates the identifiers and the relevant attributes of entities declared in other modules.

Definitions may occur in all four program areas (module, task, procedure, block), but certain definitions are only permitted in certain areas. The relationship between program areas and definitions are shown in Table 4/1.

TABLE 4/1
Which Definition is Allowed in Which Program Area?

<i>entity \ area</i>	<i>module</i>	<i>task</i>	<i>procedure</i>	<i>BEGIN-block</i>
task	X			
procedure	X			
BEGIN-block		X	X	X
local-mode	X	X	X	X
device	X			
file	X			
semaphore	X			
interrupt	X			
signal	X			

Where: local-mode (see 4.4.2) indicates variables and arrays of single-mode (see 4.2.3) and structure mode (see 4.2.4.2).

4.1 Valid Scopes of Definitions

Definition of an identifiable entity is only valid in the area in which the definition is located. This area does not contain the label of the area but includes nested inner areas. However, if an inner area contains a definition

for an identifier already defined in the outer area, the outer definition becomes invalid in the inner area (and its sub-areas), but becomes valid again when this inner area is left. The values of variables, whose identifiers were invalidated in an inner area, also become available again then.

The limited scopes of definitions have two consequences:

- Data defined in an area is not accessible outside that area (data declared at module level and provided with the GLOBAL-attribute can be regarded as being at its own level (program level), the highest level containing all program modules as inner areas).
- Jumps by GOTO or conditional-statements from one task or procedure into another task or procedure, or even from one module into another module are not permissible. In addition, a jump from an external area to a labelled statement within a BEGIN-block is illegal.

4.2 Variables

In algebra, a variable is a named entity representing a value which can be modified in equations. In this PEARL subset a variable is an entity whose value can be changed in assignments. The values which can be assigned to a particular variable are limited to the mode stated in its definition. The permissible modes which can be assigned to variables are shown in Table 4/2 in 4.3.5. The declaration

DCL W FIXED;

gives the attribute FIXED to a variable with the identifier W (i.e. it restricts values that the variable can take to integer numbers). Variables which have the same attributes (see 4.3) can be put together within parentheses (see 3.4 Identifiers).

For example:

SPC (X, Y, Z) FLOAT;

three variables, identified as X, Y, Z, are given the attribute FLOAT (i.e. they can only take real values).

Before a variable can be used within expressions a value must have been assigned to it.

4.3 Attributes

A definition of a program entity determines its identifier and all its attributes. These attributes, once specified, cannot be changed later in the program. The only exception is the INV attribute which can be changed.

4.3.1 Arrays

An array is a set of data elements of the same mode, subscripts being used to address individual elements of the array (in contrast to structures, see 4.3.2.2). The bounds of the array are listed in parentheses following the identifier:

'('INTEGER-CONSTANT //','')'.

For example, the declaration

DCL A (4, 3) FLOAT;

defines an array, A, which consists of twelve variable elements of mode FLOAT. The first subscript ranges from 1 to 4, the second from 1 to 3. Table 4/2 (in 4.3.5) shows the modes permitted for arrays (and array elements).

An array element is identified by the identifier of the array followed by a set of subscripts in parentheses:

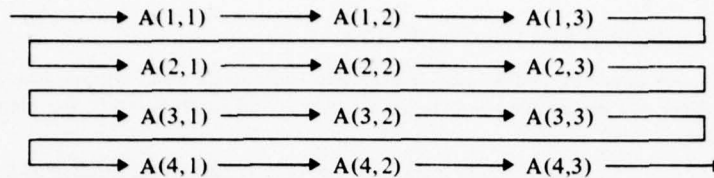
IDENTIFIER'('(EXPRESSION-SEVEN //','')'.

EXPRESSION-SEVEN (see 5.2) is an expression which may contain operators and variables. Its result must be an integer value.

Elements of multidimensional arrays are stored in the following order: first the elements referred to by the last subscript (in ascending order), then those of the last but one, etc.

Example:

The elements of the array A(4,3) are stored in memory:



A(1,2) is the second and A(2,1) the fourth element in the array.

In this subset, only array elements can be accessed, but not arrays as a whole, except as procedure parameters (see 7.2 Procedure Declaration: *PARAMETER-MODE*) and at initialization (see 4.3.3). This is in common with the concept of structures (4.3.2.2).

4.3.2 Mode Attributes

In its definition a data entity is given a single mode attribute, which determines its data type.

4.3.2.1 Modes of non-composite data

Computational constants (see 3.3) and variables, which can take such constant values, can be directly combined with operators. The modes of such data are described collectively under the syntactical term *SIMPLE-MODE*.

SIMPLE-MODE: 'FIXED' ['('INTEGER-CONSTANT')']/ 'FLOAT' ['('INTEGER-CONSTANT')']/ 'BIT' ['('INTEGER-CONSTANT')']/ 'CHAR' ['('INTEGER-CONSTANT')']/ 'DUR'/ 'CLOCK'.	mode attribute for integer data real data bit strings character strings time intervals clock time
Where: INTEGER-CONSTANT, preceded by <i>FIXED</i> or <i>FLOAT</i> , defines, for data items, possessing the mode attribute, a length, in bits, which can differ from the standard length (see 4.4.1). INTEGER-CONSTANT, preceded by <i>BIT</i> or <i>CHAR</i> , defines the number of elements in the string.	

None of the other non-composite data types can be handled directly using operators. These data items and their modes are:

<i>mode</i>	<i>characterizing key-word</i>	<i>data</i>	<i>described in para.</i>
DEVICE-MODE	'DEVICE'/'DVC'	devices	9
FILE-MODE	'FILE'	files	10
INTERRUPT-MODE	'INTERRUPT'/'IRUPT'	interrupts	9
SIGNAL-MODE	'SIGNAL'	signals	9
SEMA	'SEMA'	semaphores	17.2

4.3.2.2 Structures

A structure is a set of data elements which are normally of different modes (in contrast to arrays, see 4.3.1). There is no general mode for structures. Normally each structure attribute defines a new mode, which differs from other structure modes in the number of elements, the mode attributes of these elements, and the selector-identifiers. A selector-identifier indicates the position of an element within a certain structure. The selector-identifier uses the same format as a real identifier, but as it does not identify an entity, the same identifier can be used again to identify another entity.

STRUCTURE-MODE:
 'STRUCT' '(((IDENTIFIER/'(IDENTIFIER/'(')')')
 SIMPLE-MODE/'(')')'.

Where:

SIMPLE-MODE (see 4.3.2.1) is an arithmetic, a string, or a time mode attribute.

A structure element is identified by the structure-identifier followed by a full stop and the selector-identifier of the element's position:

IDENTIFIER.'IDENTIFIER.

In this subset structures as a whole are not addressable, except as procedure parameters (see 7.2 Procedure Declaration: PARAMETER-MODE) and at initialization (see 4.3.3). The only addressable unit is the structure element. This is in common with the concept of arrays (4.3.1).

Examples:

- 1) DCL (BL,C) STRUCT(EL1 FIXED, EL2 FLOAT, EL3 BIT(4));
 declares two structures, BL and C, with three variable elements. The first variable is of mode FIXED, the second of mode FLOAT, the third of mode BIT(4).
- 2) SPC F(5) STRUCT((N,O,P)DUR,Q CLOCK);
 defines a five-structure array, F, the first three elements of which are variables of mode DUR and the last is a variable of mode CLOCK.
- 3) F(2).N + F(4).Q
 Two of the structure elements specified in example (2) are added together (the result is of mode CLOCK).

4.3.3 INIT Attribute

Variables, arrays and structures can be given initial values when they are declared by using the INIT attribute. This consists of the key-word INITIAL (or INIT) followed by at least one EXPRESSION-SEVEN (see 5.2), which provides an initial value:

('INITIAL/'INIT')('EXPRESSION-SEVEN/'(')')'.

The value of EXPRESSION-SEVEN must be of the same mode as the data items being given the initial value.

For arrays and structures the number of EXPRESSION-SEVENs must not be greater than the number of data elements. If there are more data elements than values the surplus elements are given the value of the last entry in the list. The values of variables occurring in an EXPRESSION-SEVEN must be known when the initializing declaration is made.

Examples:

Correct initializations:

- 1) DCL CIRCUM FLOAT INITIAL(2.0* $\text{RAD} \cdot \text{PI}$);
 A variable named CIRCUM is initialized by an EXPRESSION-SEVEN.
- 2) DCL (MASK1,MASK2) BIT(5) INIT('00001'B);
 Two variables MASK1 and MASK2 are initialized.
- 3) DCL FLE(5) FLOAT INIT(-1.0,.0,1.0);
 An array FLE(5) is initialized. The surplus elements FLE(4) and FLE(5) are both given the initial value 1.0.
- 4) DCL ST1 STRUCT(N1 FLOAT, (N2,N3) BIT(3))
 INITIAL(2.71828,'101'B1,'011'B1);
 A structure which has three variables as elements is preset.
- 5) DCL ST2 STRUCT(M1 FLOAT, (M2,M3)BIT(3), M4 BIT(5))
 INIT(FLE(3), ST1.N2,ST1.N2, MASK1);
 A structure ST2 is initialized. Its first element is initialized by the element of an array (declared in example 3), its second and third element is initialized by an element of another structure (declared in example 4) and its fourth element is initialized by a variable (initialized in example 2).
- 6) DCL (MASK3,MASK4) BIT(4) INIT('1111'B,'0110'B);

4.3.4 INV Attribute

Variables, arrays and structures can be protected against modification by the attribute INV (denoting invariable) written in front of the mode attribute. Such data can be regarded as constants, arrays or structures with constant elements.

An entity containing the INV attribute in its declaration must also have the INIT attribute (see 4.3.3) unless the entity is a formal parameter (see 7.4).

Examples:

Correct use of the INV attribute:

- 1) DCL PI INV FLOAT INIT(3.14159);
- 2) DCL MASK(3) INV BIT(8) INIT('10111010'B, '01110101'B, '01010001'B);
- 3) DCL STI INV STRUCT(N1 FLOAT, (N2, N3) BIT (3)) INIT(1.486772, '110'B, '011'B);
- 4) SPC PI INV FLOAT;

Incorrect use of the INV attribute:

- 5) DCL STVI STRUCT(E1 FLOAT, E2 INV CHAR(3) INIT('AB1'));
A composite entity must not have both variable and invariable elements, i.e. elements which are to remain unchanged are not protected against modification, if variable elements are also present.
- 6) TEI:TASK:DCL MESSAGE INV CHAR(6);
.
.
.
MESSAGE = 'DANGER';
.
.
.
END; /*TEI*/
MESSAGE must be initialized (using the INIT attribute) in its declaration and the assignment statement must be excluded.

4.3.5 Relations between Modes and Entities

Only certain modes are permissible for certain entities. This is illustrated in the table below.

TABLE 4/2
Modes and Entities

		<i>scalars</i>		<i>arrays</i>		<i>elements of</i>	
		<i>INV</i>	<i>var.</i>	<i>INV</i>	<i>var.</i>	<i>INV</i>	<i>variable structures</i>
<i>modes</i>	FIXED	X	X	X	X		
	FLOAT	X	X	X	X		
	DUR	X	X	X	X		
	CLOCK	X	X	X	X		
	BIT(n)	X	X	X	X		
	CHAR(n)	X	X	X	X		
	DEVICE			one index only			
	FILE						
	SEMA		X				
	INTERRUPT				one index only		
	SIGNAL				one index only		
	STRUCT	X	X	X	X		

Devices, files, interrupts and signals are invariable, but the INV attribute is implicit (i.e. not required in their definition). At arrays and structures the INV attribute is in common for all elements and written in front of the mode attribute.

The elements of one structure are normally not all of the same mode (indicated by distorted cross lines).

4.3.6 IDENT Attribute

The IDENT (IDENTICAL) attribute permits the declaration of another identifier for a previously defined variable or structure, or an element of an array or structure. The IDENT attribute is written after the new identifier and the mode attribute (which in this case is redundant) and has the form:

('IDENTICAL'/'IDENT')('SYMBOL')

where SYMBOL represents the whole identifying name already defined earlier:

SYMBOL: IDENTIFIER	denotes a variable or structure	see
('IDENTIFIER/ '('EXPRESSION-SEVEN'/'')')\$.	a structure element	4.3.2.2
	an array element	4.3.1

Example: DCL V FLOAT IDENTICAL(W);

The identifier V is introduced for the already defined variable W.

If the new identifier is provided with a set of subscripts, i.e. a full identifying name of an array element is declared, the subscripts are considered as the upper bounds of a new array. It must be noted that the new array must comprise of elements of the same mode.

Examples:

Correct use of the IDENT attribute:

- 1) DCL MIN FLOAT IDENT(BL.EL2);
A structure element is declared as a variable with the identifier MIN.
- 2) DCL SA FLOAT IDENT(A(4,3));
An array element is declared as a variable with the identifier SA.
- 3) DCL SF DUR IDENTICAL(F(2).P);
An element of a structure, which in turn is an element of an array, is declared as a variable, SF, (see 4.3.2.2, example 2).

Incorrect use of the IDENT attribute:

- 4) DCL K.K2 FLOAT IDENT(BL.EL2);
It is not legal to declare structure elements, since the beginning of the structure is undefined.

4.3.7 GLOBAL-Attribute

Entities used in more than one module must be defined in all modules in which they occur (in one module by declaration, in all others by specification). The definitions for such entities must be at module level and each must contain the GLOBAL-attribute. Such entities can be data, tasks, or procedures. In the definitions of data items and in the definition of task and procedure headings, the GLOBAL-attribute comes last unless the RESIDENT attribute is present. A global qualifier contained in a GLOBAL-attribute implicitly defines a library which contains the entities concerned. If no global qualifier is specified the library is either defined by the global qualifier attached to the key word MODULE, or is a general library.

GLOBAL-ATTRIBUTE:
'GLOBAL'['(GLOBAL-QUALIFIER)'].

Where:

GLOBAL-QUALIFIER is the identifier of a library.

4.3.8 RESIDENT Attribute

Provided there is a back-up storage device and an overlay facility then the program will normally be kept in the back-up device. Parts of the program are transferred into core for execution and remain there until they are no longer required. However, frequently used program entities can be core-resident all the time in order to save the transfer times required to bring the various entities into core. Such entities must be indicated by using the RESIDENT attribute in their declarations. Only global entities (see 4.3.7) can be given the RESIDENT attribute which consists solely of the key-word RESIDENT. The RESIDENT attribute is not written in specifications. If there is no back-up store available the RESIDENT attribute becomes meaningless.

4.4 Definitions at Module Level

At module level (or more exactly, in the problem area) definitions only are allowed. Their order is fixed as:

```
'MODULE' .
.
.
'PROBLEM';
(LENGTH-DECLARATION';')$
(GLOBAL-SPECIFICATION';')$
(MODULE-IDENTIFIER-DECLARATION';')$
(LABEL-IDENTIFIER':'(PROCEDURE-DECLARATION/
TASK-DECLARATION)';')$
'MODEND';
```

Where:

LENGTH-DECLARATION defines the standard lengths of integer and real data in the actual module,

GLOBAL-SPECIFICATION specifies all entities declared in other modules and needed in the actual module,

MODULE-IDENTIFIER-DECLARATION declares all data used in more than one task or procedure,

PROCEDURE-DECLARATION is described in 7.2,

TASK-DECLARATION is described in 14.1.

4.4.1 Declaration of Standard Lengths

The standard lengths of integer and real data are defined in one declaration.

```
LENGTH-DECLARATION:
'LENGTH' (('FIXED'/'FLOAT')('INTEGER-CONSTANT')//',').
```

Where:

INTEGER-CONSTANT represents the length expressed in bits.
In case of FLOAT it defines the length of the mantissa.

Example: LENGTH FIXED(16), FLOAT(32);

4.4.2 Specifications

Specifications are allowed at module level only. All explicitly definable entities except BEGIN-blocks can be specified provided they are declared within other modules. If these modules are written in PEARL the corresponding entities must be given GLOBAL-attributes there. The order of the specified entities is arbitrary.

```
GLOBAL-SPECIFICATION:
('SPECIFY'/'SPC') (('IDENTIFIER'/'('IDENTIFIER'//',')')
(LLOCAL-MODE/
'SEMA'/
['('INTEGER-CONSTANT')'] ('INTERRUPT'/'IRUPT')/
['('INTEGER-CONSTANT')'] 'SIGNAL'/
['('INTEGER-CONSTANT')'] ('DEVICE'/'DVC')DEVICE-TYPE/
'FILE' [FILE-TYPE]/
PROCEDURE-MODE/
'TASK')
['GLOBAL' ['('GLOBAL-QUALIFIER')']] //',').
```

Where:

IDENTIFIER is the name of an entity being specified (see 3.4),
 LOCAL-MODE is a syntactical abbreviation for a set of attributes (see below),
 INTEGER-CONSTANT represents the upper bound of an array,
 PROCEDURE-MODE represents the characteristics of a procedure (see 7.3),
 GLOBAL-QUALIFIER denotes a library (see 4.3.7).

LOCAL-MODE:
 ['(INTEGER-CONSTANT//','')'] ['INV']
 (SIMPLE-MODE/
 STRUCTURE-MODE).

see
 INV in 4.3.4
 4.3.2.1
 4.3.2.2

Where:

INTEGER-CONSTANT represents the upper bound of an array.

LOCAL-MODE data comprises invariable and variable scalars and arrays of SIMPLE-MODE and STRUCTURE-MODE. LOCAL-MODE can describe certain data in declarations (see 4.4.3, 4.5) and specifications and the form of a file (FILE-CONTENTS, see 10.2).

Example:

Correct specifications:

- 1) SPECIFY PI INV FLOAT,
 AB(3,4) FLOAT,
 ATT(3) SIGNAL, FAIL SEMA.
 MAN STRUCT((M1,M2) FLOAT, M3 CHAR(6));

Incorrect specifications:

- 2) SPC PI INV FLOAT INIT(3.14159);
 INIT (and IDENT) attribute is not allowed in specifications.

4.4.3 Declaration of Data at Module Level

Data used in more than one task or procedure must be declared at module level. If any of the tasks or procedures using the data lies in another module, the GLOBAL-attribute must be attached to each data item concerned, in its declaration. Data declarations at module level are similar in format to specifications. The INIT and IDENT attributes are only permitted in declarations. The RESIDENT attribute can occur only in declarations which have been given the GLOBAL-attribute. Procedures and tasks have special forms of declarations (see 7.2, 14.1).

MODULE-IDENTIFIER-DECLARATION:
 ('DECLARE'/'DCL') ((IDENTIFIER/'(IDENTIFIER//','')')
 (LOCAL-MODE [(INITIAL/'INIT') ('(EXPRESSION-SEVEN//','')')/
 ('IDENTICAL/'IDENT')SYMBOL]/
 'SEMA'/
 ['(INTEGER-CONSTANT')'] ('INTERRUPT'/'IRUPT')
 ['(INTEGER-CONSTANT')'] 'SIGNAL'/
 ['(INTEGER-CONSTANT')'] ('DEVICE'/'DVC')DEVICE-TYPE/
 'FILE' [FILE-TYPE])
 ['GLOBAL' ['(GLOBAL-QUALIFIER')'] ['RESIDENT']] //',').

Where:

IDENTIFIER denotes a data item being Declared,
 LOCAL-MODE is described in 4.4.2 and may be given the INIT or IDENT attribute (see 4.3.3, 4.3.6).
 INTEGER-CONSTANT gives the upper boundary of an array.

Examples:

Correct declarations:

- 1) DECLARE (C,U,D) DUR GLOBAL RESIDENT;
- 2) DCL E CLOCK GLOBAL(COMMON),
 F BIT(8);

- 3) Example (2) can also be written as:
DCL E CLOCK GLOBAL (COMMON);
DCL F BIT(8);
- 4) DCL G INV CHAR(5) INIT('WAIT '),
(M,N) DEVICE,
L SIGNAL GLOBAL,
K(5) INTERRUPT,
Q FLOAT INITIAL (7.358942),
DT FILE;
- 5) DECLARE (L,M)(4) CLOCK;
Two arrays, each containing four elements, are declared.

Incorrect declarations:

- 6) DCL (L(4),M(4)) CLOCK;
Only identifiers may form a list. Corrected in (5).

4.5 Declarations at Task, Procedure and BEGIN-Block Level

Declarations of tasks, procedures and BEGIN-blocks consist of a declaration heading and a block-tail. Whilst the heading has a specific form for each of these three declarations, the block-tail is constructed in the same way in each case and can obtain subdeclarations at its beginning. Such subdeclarations are the declarations proper at task, procedure or BEGIN-block level and can be summarized as block level declarations. Their syntactical term is LOCAL-IDENTIFIER-DECLARATION.

```
LOCAL-IDENTIFIER-DECLARATION;
('DECLARE'/'DCL') ((IDENTIFIER/'(' (IDENTIFIER //','))' )
LOCAL-MODE [( 'INITIAL'/'INIT' )
              ('EXPRESSION-SEVEN //',')' )/
              ('IDENTICAL'/'IDENT') SYMBOL] //',' ).
```

Where:

IDENTIFIER denotes the data item being declared.

LOCAL-MODE is described in 4.4.2 and may be given the INIT or IDENT attribute (see 4.3.3, 4.3.6).

All variables, contained within an initializing declaration are re-initialized each time the block is re-entered.

Examples:

Correct declarations at block level:

```
1) T1: TASK;
   DCL CL1 CLOCK INIT(4:0:30),
     STA STRUCT (N1 FLOAT, N2 BIT(3)),
     BS2 BIT(2) INIT('01'B),
     DE INV DUR INIT(7MIN 1SEC 31MSEC);
     .
     .
     .
   BEGIN;
     DCL TS CLOCK INIT(DE+CL1+100MSEC),
     BN(4) BIT(5) INIT('101'B<>BS2),
     WEIGHT FLOAT;
     .
     .
     .
   END;
     .
     .
     .
   END;/* T1 */
```

Incorrect declarations at block level:

- 2) DCL PI INV FLOAT INIT(3.14159)GLOBAL;
The GLOBAL- (and RESIDENT) attribute is not permitted at block level.

3) DECLARE SYNC SEMA,15(10)IRUPT,
DE(6) DEVICE, FE FILE, SE SIGNAL;

Declaration of semaphores, interrupts, devices, files and signals is not permitted at block level.

5. OPERATORS AND EXPRESSIONS

An expression is an algorithm used for computing a required value. An expression is made up of a sequence of characters, consisting of operands, operators and parantheses. A single item (e.g. a variable or a constant) can also be an expression.

Examples: $(X + 3) * Y + B(I, K, J * 3)$
K
A(5)
 $-1025 + Z$

5.1 Operators

Operators specify the kind of operation to be performed upon one or two operands.

5.1.1 Monadic Operators

Monadic operators require a single operand only. They appear only at the beginning of an expression, or after a left-hand bracket.

MONADIC-OPERATOR:
'+'/'-'/'NOT'/'FIX'/'FLOAT'/'BIT'/'CHAR'.

Examples: -5 '-' is a monadic operator
 $+A(I, J)$ '+' is a monadic operator
 $B1 \text{ AND } (\text{NOT } C)$ 'NOT' is a monadic operator

5.1.2 Dyadic Operators, Order of Precedence

Dyadic operators require both a left and a right operand. They are classified according to their order of precedence.

PRECEDENCE-SEVEN-OPERATOR:	'OR'.
PRECEDENCE-SIX-OPERATOR:	'AND'.
PRECEDENCE-FIVE-OPERATOR:	'=='/'EQ'/' '/'/'NE'.
PRECEDENCE-FOUR-OPERATOR:	'<'/'LT'/' '>'/'GT'/' '<='/'LE'/' '>='/'GE'.
PRECEDENCE-THREE-OPERATOR:	'+'/' '-'/' '<>'/'CAT'/' '><'/'CSHIFT'/' 'SHIFT'.
PRECEDENCE-TWO-OPERATOR:	'*'/' '/'.
PRECEDENCE-ONE-OPERATOR:	'**'/' 'FIT'.

The order of execution of an expression is determined by the parantheses within it, and by the precedence of its operators according to the following rules:

- parantheses enclosing an expression mean that the parenthesised expression is to be executed before the operator preceding the opening paranthesis or the operator following the closing paranthesis is performed.

– if there are no parantheses within an expression the order of execution is given by the precedence of the operators:

- operations with higher precedence operators are operated first
- if an expression contains operators of the same precedence (levels 2 to 7), the left most operator is performed first
- if an expression contains operators with the same precedence (level 1 and monadic operators), the expression is calculated from right to left.

Examples:

$-A+B+C*D/E$

step 1: $Z1 = -A+B$

step 2: $Z2 = C*D$

step 3: $Z3 = Z2/E$

step 4: $Z4 = Z1+Z3$

$-A**B**C$

step 1: $Z1 = B**C$

step 2: $Z2 = A**Z1$

step 3: $Z3 = -Z2$

5.2 Expressions

The following syntactical rules must be observed in the composition of expressions:

EXPRESSION-SEVEN:

EXPRESSION-SIX // PRECEDENCE-SEVEN-OPERATOR.

EXPRESSION-SIX:

EXPRESSION-FIVE // PRECEDENCE-SIX-OPERATOR.

EXPRESSION-FIVE:

EXPRESSION-FOUR // PRECEDENCE-FIVE-OPERATOR.

EXPRESSION-FOUR:

EXPRESSION-THREE // PRECEDENCE-FOUR-OPERATOR.

EXPRESSION-THREE:

EXPRESSION-TWO // PRECEDENCE-THREE-OPERATOR.

EXPRESSION-TWO:

EXPRESSION-ONE // PRECEDENCE-TWO-OPERATOR.

EXPRESSION-ONE:

PRIMITIVE-EXPRESSION

PRECEDENCE-ONE-OPERATOR EXPRESSION-ONE /
MONADIC-OPERATOR EXPRESSION-ONE.

PRIMITIVE-EXPRESSION:

CONSTANT-DENOTATION / SYMBOL /
'(EXPRESSION-SEVEN)'.

The following operands may be used in the construction of expressions (according to the rules outlined above):

constants
variables
variable array elements
structure elements
function calls
expressions in parantheses.

5.2.1 Arithmetic Operations

The following table lists the permissible operand and result modes for arithmetic operations:

operator	mode of		result
	left operand	right operand	
+ (monadic)	—	FIXED	FIXED
	—	FLOAT	FLOAT
	—	DUR	DUR
	—	CLOCK	CLOCK
— (monadic)	—	FIXED	FIXED
	—	FLOAT	FLOAT
	—	DUR	DUR
**	FIXED FLOAT	FIXED FLOAT	FIXED FLOAT
*	FIXED	FIXED	FIXED
	FLOAT	FLOAT	FLOAT
	DUR	FIXED	DUR
	FIXED	DUR	DUR
/	FIXED FLOAT	FIXED FLOAT	FIXED FLOAT
+	FIXED	FIXED	FIXED
	FLOAT	FLOAT	FLOAT
	DUR	DUR	DUR
	DUR	CLOCK	CLOCK
—	CLOCK	DUR	CLOCK
	FIXED	FIXED	FIXED
	FLOAT	FLOAT	FLOAT
	DUR	DUR	DUR
	CLOCK	DUR	CLOCK

5.2.2 Logical Operations

Logical operators operate on bit strings of equal length.

operator	mode of		result
	left operand	right operand	
NOT		BIT(n)	BIT(n)
AND	BIT(n)	BIT(n)	BIT(n)
OR	BIT(n)	BIT(n)	BIT(n)

The operations are performed on a bit-by-bit basis. As a result of the operations, each bit position is given the value defined in the following table:

A	B	NOT A	NOT B	A AND B	A OR B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

Examples:

A = '010110 B

B = '111111'B

C = '101001'B

A AND B OR C has the result '111111'B

A AND (B OR C) has the result '010110'B

5.2.3 Comparison Operations

The comparison operators included in the PEARL-Subset are:

EQ	or	==
NE	or	/=
LT	or	<
GT	or	>
LE	or	<=
GE	or	>=

There are four types of comparisons:

- algebraic, i.e. the comparison of numeric values
- character, i.e. the comparison of two character strings of the same length
- bit, i.e. the comparison of two bit strings of the same length
- time, i.e. the comparison of clock and duration values.

The operand modes permitted for the different comparison operators are given in the following table. A comparison results in a bit being set or cleared: The value is set to '1'B, if the relationship is true and '0'B, if the relationship is false.

operator	mode of		result
	left operand	right operand	
GT or >	FIXED	FIXED	BIT(1)
LT or <	FLOAT	FLOAT	
GE or >=	DUR	DUR	
LE or <=	CLOCK	CLOCK	
EQ or ==	FIXED	FIXED	BIT(1)
NE or /=	FLOAT	FLOAT	
	DUR	DUR	
	CLOCK	CLOCK	
	BIT(n)	BIT(n)	
	CHAR(n)	CHAR(n)	

Examples: A LE 10
A+B GT B
A GT B EQ 1 LT K

5.2.4 Concatenation Operations

The concatenation operator operates on bit strings or character strings. The result of a concatenation operation is a bit or character string, the length of which is the sum of the lengths of the two operands.

operator	mode of		result
	left operand	right operand	
CAT or <>	BIT(n)	BIT(m)	BIT(n+m)
	CHAR(n)	CHAR(m)	CHAR(n+m)

Examples:

A = '1010'B
 B = '11'B
 C = 'XY,Z'
 D = 'AA/BB'

A CAT B CAT '00'B has the result '10101100'B
 C CAT D has the result 'XX,ZAA/BB'

5.2.5 Shift Operations

Shift operations operate on bit strings. The left operand designates the shift operand, the right operand the number of places to be shifted. If the right operand is negative, a right shift is performed, otherwise the shift is to the left.

operator	mode of		result
	left operand	right operand	
SHIFT CSHIFT or ><	BIT(n)	FIXED	BIT(n)

The operator SHIFT denotes a logical shift, and operator CSHIFT or >< a cyclic shift.

Examples:

A = '101010'B
 B = '110110'B

A SHIFT (-2) has the result '001010'B
 B CSHIFT (-2) has the result '101101'B
 A CAT B SHIFT 6 has the result '110110000000'B

5.2.6 Conversion Operations

Conversion operators perform the conversion of an operand from one mode to another.

operator	mode of		result
	left operand	right operand	
FLOAT	—	FIXED	FLOAT
FIX	—	BIT(n)	FIXED
CHAR	—	INT	CHAR(1)
BIT	—	FIXED	BIT(n)
FIT	FIXED	FIXED	FIXED

The various conversions performed by the above operators are described in the following list:

FLOAT: Conversion of a FIXED operand to a FLOAT operand of standard length
 FIX: Conversion of a bit string to a FIXED number of standard length
 CHAR: Conversion of a FIXED value to a character
 BIT: Conversion of a FIXED value to a bit string of the same length
 FIT: The operator changes the length of the FIXED value. The right operand indicates the length, the left operand is to be converted to.

6. STATEMENTS

A statement is the basic processing unit of a program. Statements are used in the handling of information and in the control of program execution. They are executed in the sequence in which they are written, unless a different order is demanded by the control of program execution. They consist of an optional identification prefix followed by the statement body.

STATEMENT:
 ((LABEL-IDENTIFIER ':')\$/
 ('ON' SIGNAL-IDENTIFIER ['('INTEGER-CONSTANT')'] ':')\$)
 STATEMENT-BODY ';'.

Where:

LABEL-IDENTIFIER denotes the statement,

SIGNAL-IDENTIFIER represents a signal,

INTEGER-CONSTANT represents an element of a one-dimensional array of signals.

A label-identifier may occur as the destination of a GOTO-statement, or be used, merely to enhance the clarity of the program.

Using the ON prefix, the execution of a statement can be made to be dependent upon the occurrence of a signal. A signal is generated on the occurrence of an illegal task operation, e.g. overflow or divide fault. (For comparison: an interrupt is not generated by the program itself but by the process.) In a similar way to interrupts, a signal identifier is introduced and associated with a particular class of illegal operations in the system division. The definition of a signal is made in the problem division at module level and must include the SIGNAL attribute. Unlike interrupts, a signal is only communicated to the task which performed the operation causing the signal.

STATEMENT-BODY:	described in
ASSIGNMENT /	6.1
BEGIN-BLOCK /	6.2
SEQUENTIAL-CONTROL /	6.3
PARALLEL-CONTROL /	16.
SYNCHRONISATION /	17.3
INTERRUPT /	15.2
TRANSPUT .	Part III

Execution control is performed by the sequential-control, the parallel-control and the ON prefix just mentioned, and is affected by the synchronisation and interruption.

The information processing is carried out for the most part within assignment statements.

6.1 Assignment

A symbol can be given the value of an expression-seven by using the ASSIGNMENT-statement. A symbol, in this context, can be variable or an element of an array or structure (see SYMBOL in 4.3.6).

ASSIGNMENT:
 SYMBOL ('='/'='') EXPRESSION-SEVEN.

If the mode of the expression is different to the mode of the symbol, a conversion from the mode of the expression-seven to the mode of the symbol may be made. The possible conversions are listed in the table below and supplement the facilities of the conversion operators.

case	symbol = expression-seven
1	FIXED ← FLOAT
2	BIT(n) ← FLOAT
3	BIT(m) ← BIT(n)
4	CHAR(m) ← CHAR(n)

Case 1: FIXED ← FLOAT

The value of the expression-seven is rounded down to the nearest integer number. If the value is negative the absolute value is truncated. The conversion is limited by:

$$\text{negmax} \leq \text{FLOAT value} < \text{posmax} + 1$$

Where: negmax and posmax signify the smallest negative and the greatest positive integer number, respectively.

If negative numbers are represented by the computer in 2's complement form the above limit becomes:

$$-2^{f-1} \leq \text{FLOAT value} < 2^{f-1}$$

Where: f is the standard length of integer numbers expressed in bit positions.

Examples:

1) $f = 8$: $-128 \leq \text{FLOAT value} < 128 = 2^7$

symbol	expression-seven
-30	-3E1
- 7	-7.8
+ 7	+7.8
122	1.2256E2

2) LENGTH FIXED(16), FLOAT(32);

DCL (H,J) FIXED,

K FLOAT INIT(8.357E3),

L FLOAT INIT(4.76E4);

.

.

.

H = K;

J = L;

The second assignment is incorrect as the converted value of L (47600) is greater than $2^{15} = 32768$.

Case 2: BIT(n) \leftarrow FLOAT

The conversion is split up internally into:

- a conversion from FLOAT to FIXED (the restrictions of case 1 are again valid).
- a transformation from the computer-internal FIXED representation into 2's complement form if the internal representation is different from the 2's complement (this is done in order to maintain machine independence).
- a conversion of the integer number (in 2's complement) into a bit-string.

The length, n , of the bit-string can be calculated as follows:

$$\text{FIXED value} > 0: 2^{n-2} \leq \text{FIXED value} < 2^{n-1}$$

$$\text{FIXED value} < 0: 2^{n-2} < \text{FIXED value} \leq 2^{n-1}$$

Examples:

attribute	symbol	expression-seven	
	bit-string \leftarrow FIXED value \leftarrow FLOAT value		
BIT(4)	'0100'B	4	4.2
BIT(3)	'100'B	-4	-4.2
BIT(11)	'0100000011'B	515	5.156E2

Cases 3 and 4: BIT(m) \leftarrow BIT(n)

CHAR(m) \leftarrow CHAR(n)

The contents of the expression-seven is assigned, left-justified, to the symbol.

For $n < m$ the free positions are filled with zeroes in the case of bit-strings and with spaces in the case of characters.

For $n > m$ the surplus positions are dropped.

Examples:

symbol	m ← n		expression-seven
'101100'B	BIT(6)	BIT(4)	'1011'B
'1011'B	BIT(4)	BIT(6)	'101101'B
'TEST 24'	CHAR(7)	CHAR(11)	'TEST 24 RIG'

6.2 BEGIN-Block

A BEGIN-block is a sequence of declarations and statements introduced by the key-word BEGIN and terminated by the key-word END.

The advantage of forming a block is that the core occupied by entities defined in the block becomes available to entities outside the block when the block is not being executed (common storage).

```

BEGIN-BLOCK:
'BEGIN' ';' BLOCK-TAIL.

BLOCK-TAIL:
(LOCAL-IDENTIFIER-DECLARATION';')$
STATEMENT $
'END'.

Where:
LOCAL-IDENTIFIER-DECLARATION means declaration of
local-mode data.

```

Block-tails also constitute the main parts of tasks and procedures. From the fact that a BEGIN-block (in more general terms, a block-tail) is considered self-contained it follows that definitions (labels as well as local-mode data) made within a block-tail are not valid outside the block-tail. All the restrictions stated in Chapter 4.1 are applicable to block-tails.

Example:

```

Incorrect jump into a block:

BT: TASK;
    DCL(A,BB) FLOAT;
    .
    .
    GOTO EB1;
BEGIN;
    DCL AB FLOAT;
EB1:  AB = A-BB;
    .
    .
    .
    END;
END;

```

A jump into a block-tail is not possible as a block internal label (in the example, EB1) is not known outside the block. The declarations of a BEGIN-block are non-executable and therefore there is no need to use a GOTO-statement in the program in order to by-pass them.

6.3 Sequential-Control

Sequential-control statements determine the path of execution *within* tasks, procedures, and between tasks and procedures. This is in contrast to parallel-control statements which determine the path *between* tasks only and always act through the operating system.


```

SEQUENTIAL-CONTROL:
SKIP-STATEMENT/
GOTO-STATEMENT/
CONDITIONAL-STATEMENT/
CASE-STATEMENT/
REPEAT-STATEMENT/
CALL-STATEMENT/
RETURN-STATEMENT.

```

6.3.1 SKIP-Statement

This statement is a dummy statement and does not change the path of execution.

It may be used temporarily and can be replaced later by an executable statement.

```

SKIP-STATEMENT:
'SKIP'.

```

6.3.2 GOTO-Statement

Normally the statements are executed in the order in which they are written. The GOTO-statement permits deviation from this order by performing an unconditional jump to the statement, the name of which is stated in the GOTO-statement.

```

GOTO-STATEMENT:
'GOTO' LABEL-IDENTIFIER.

Where:
LABEL-IDENTIFIER is the name of a statement.

```

A jump from outside into a composed entity terminated by END (task, procedure, BEGIN-block, REPEAT-statement) or by FIN (conditional-statements, CASE-statement) is not allowed as such entities are considered self-contained.

An exit from the block-tail of a task or procedure using GOTO is also not permitted.

6.3.3 Conditional-Statement

Using the conditional-statement the path of execution can be varied according to the logical value (true = '1'B and false = '0'B) of a condition.

```

CONDITIONAL-STATEMENT:
'IF' BIT-ONE-EXPRESSION-SEVEN
'THEN' STATEMENT*
['ELSE' STATEMENT*] 'FIN'.

Where:
BIT-ONE-EXPRESSION-SEVEN is an expression of mode BIT(1).
Together with IF it forms a condition.

```

If the condition is satisfied (true) then the branch comprising the statement(-sequence) between THEN and ELSE or, if ELSE is omitted, between THEN and FIN, is executed. If the condition is false the branch between ELSE and FIN is executed. If the latter branch does not contain statements ELSE is omitted. After executing one of the two branches control will return to the statement after FIN, unless the branch taken contained a GOTO-statement which passed control beyond the FIN.

There are no restrictions on the branch statements, so that any sequential-control-statement (e.g. multiply nested conditional-statements) can be used.

6.3.4 CASE-Statement

Using the CASE-statement the path of execution can be split into several branches, according to the integer value of an expression. The statement provides a shorthand method of writing certain conditional-statements.

CASE-STATEMENT:
 'CASE' INTEGER-EXPRESSION-SEVEN
 ('ALT' STATEMENT \$)*
 ['OUT' STATEMENT *] 'FIN'.

Where:

INTEGER-EXPRESSION-SEVEN is an expression delivering a value of mode FIXED.

If the value of the expression is *n* control is passed to the *n*-th branch. The branch begins with the *n*-th ALT and ends with the next ALT, or if there is no more ALT, with OUT, or if OUT does not exist, with FIN. A particular branch need not contain statements. If *n* is negative or greater than the number of alternative branches, the branch containing the statement(-sequence) between OUT and FIN is executed. If the latter branch does not contain any statements OUT is omitted. After performing one of the branches the statement after FIN is executed, unless the branch taken contained a by-passing GOTO-statement.

There are no restrictions on the branch statements so that any sequential-control-statement (e.g. nested CASE-statements) may occur.

6.3.5 REPEAT-Statement

The REPEAT-statement is used if it is required to execute a set of instructions several times (i.e. program looping). It consists of the loop statement(-sequence) and a preceding clause comprising a loop count mechanism and/or a condition.

REPEAT-STATEMENT:
 ['FOR' IDENTIFIER
 + 'FROM' EXPRESSION-SEVEN
 + 'BY' EXPRESSION-SEVEN
 + 'TO' EXPRESSION-SEVEN
 + 'WHILE' EXPRESSION-SEVEN]
 'REPEAT' STATEMENT * 'END'.

Where:

IDENTIFIER	represents a loop counter which is declared by the REPEAT-statement and is only valid within it. If the term 'FOR' IDENTIFIER is omitted the loop counter is not accessible.
EXPRESSION-SEVEN	preceded by 'FROM' signifies the initial value of the loop counter. If this term is omitted the initial value is assumed to be 1.
EXPRESSION-SEVEN	preceded by 'BY' signifies the step width by which the loop counter is changed after each execution of the loop statement(-sequence). If the term is omitted the step width is assumed to be 1.
EXPRESSION-SEVEN	preceded by 'TO' represents the final value of the loop counter. When the counter exceeds this value the iteration process is terminated.
EXPRESSION-SEVEN	together with 'WHILE' represents a condition which must deliver a value of mode BIT(1). If the condition is false the iteration process is terminated, otherwise the loop statement(-sequence) is executed. If the term is omitted its value is assumed to be '1'B (= true).
STATEMENT *	represents the loop statement(-sequence).

N.B.: The initial value, the step width and the final value must be integer values, which may be negative.

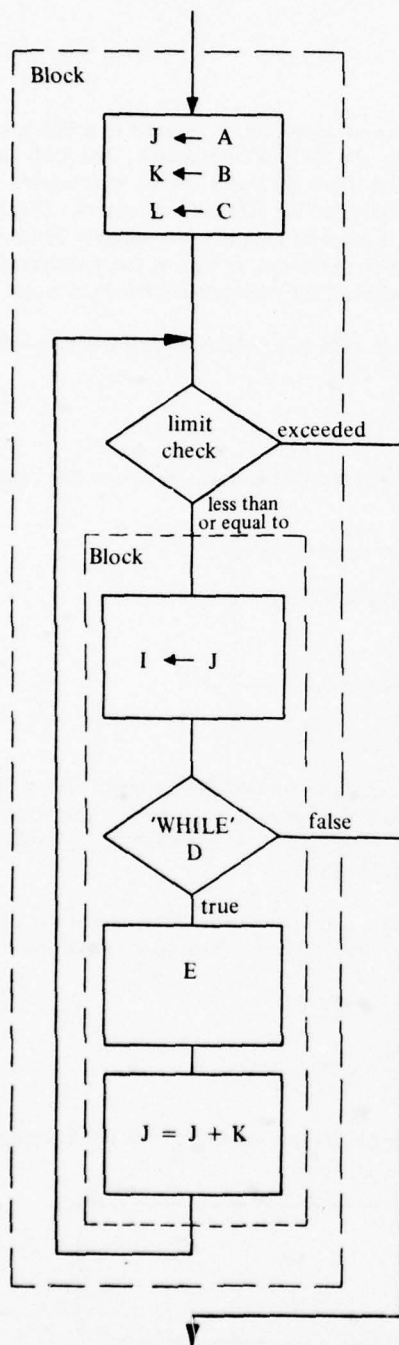
The order of execution of the REPEAT-statement

```

FOR I FROM A BY B TO C WHILE D
  REPEAT
    E;
  END;

```

is illustrated in the following flow-chart.



First a block is entered. In it a loop variable J and two invariable entities K and L are declared and initialized by A , B , and C respectively.

In the limit check J is examined, if it exceeds the final value

i.e. $J \geq L$ for $K > 0$

$J \leq L$ for $K < 0$.

If not, an inner block is entered. In it the loop counter I is declared, defined as invariable and preset by J .

If the WHILE condition D is false the statement following the REPEAT-statement is executed next.

If D is true the statement(-sequence) E is executed.

J is incremented and a jump back to the limit check is performed.

The effects of the REPEAT statement can be explained by considering the following piece of program (for simplification the final value C is omitted, i.e. the limit check is not considered):

```

BEGIN;
  DCL J FIXED INIT(A), K INV FIXED INIT(B);
M:BEGIN;
  DCL I INV INIT(J);
  IF D THEN
    E;
    J=J+K;
    GOTO M;
  FIN;
END;
END;

```

Summary:

The loop counter, I, is declared in the REPEAT-statement. So an identifier, I, defined in a block which contains the REPEAT-statement is made temporarily inaccessible by the REPEAT-statement. The loop counter I must not be used in A, B, and C. Its value must not be changed in D or E, but it can be interrogated there. A, B, and C are evaluated only once at the beginning of the execution of the REPEAT-statement. If the value of I is to be interrogated after execution of the REPEAT-statement, it must be assigned to a variable which is accessible both inside and outside the REPEAT-statement. The whole REPEAT-statement, as well as the statement(-sequence), E, acts like a BEGIN-block. Therefore a jump using GOTO to a labelled statement inside the loop is not allowed.

There are no restrictions on statements in the loop so that (e.g. calls of procedures, GOTO-statements) branching to destinations outside the loop and nesting of loops are permitted.

6.3.6 CALL-Statement

Procedures, other than function procedures, must be called using the CALL-statement. For this reason they are termed call procedures.

CALL-STATEMENT:
 'CALL' IDENTIFIER {'('EXPRESSION-SEVEN//','Y')'}.

Where:
 IDENTIFIER is the name of the call procedure.
 EXPRESSION-SEVEN represents a parameter.

Data is transferred to the called procedure via parameters included in the CALL-statement. These parameters are classified as "actual" parameters, and must be listed in the CALL-statement in a one-to-one correspondence with the parameters (classified as formal) listed in the called procedure. The formats of both lists must be identical, that is arrays must be of the same size, parameters must be of the same mode etc.

The procedure heading is the only point at which the procedure can be entered. After execution of the procedure has been completed control is returned to the statement immediately following the CALL-statement.

Examples: CALL CPE1(2.5,'101'B,DATA(10));
 CALL CPE2(X-A/H*4,'ITEM 4');

6.3.7 RETURN-Statement

On completion of a procedure execution the RETURN-statement returns control to the point logically following the procedure call in the calling task or procedure.

RETURN-STATEMENT:
 'RETURN' ['('EXPRESSION-SEVEN')'].

Where:
 EXPRESSION-SEVEN is obligatory when function procedures are used.
 It contains the result of the function procedure.

For function procedures, the result of the expression-seven must match the mode of the result-attribute stated in the heading of the procedure definition. On execution of the RETURN-statement the result is returned to the calling point which is an operand in an expression.

For call procedures, a RETURN-statement must not contain an expression-seven, and may be omitted if it is followed by the END of the procedure block-tail.

The RETURN-statement can only occur in procedures. GOTO-statements cannot be used to exit from a procedure.

Examples: RETURN(ALPHA*DIST/M/2.85*N);
RETURN(RESULT);

7. PROCEDURES

The formulation of a lengthy sequence of statements as a procedure is advantageous if the sequence is required more than once in a program. Using procedures, core is saved, as the sequence must be included only once in the program. The clarity of the program is enhanced as repetitions of the sequence are replaced by simple procedure calls. Procedures needed to solve commonly occurring problems, should be written in a general way, so that they can be collected into a "library of procedures", and so reduce the amount of programming required in future programs.

Flexibility and independence from the actual data of a general problem procedure is achieved by transferring information, in the form of parameters, between the procedure calls and the procedure. The parameters representing the information in the procedure calls are termed actual, the parameters receiving the information in the procedure itself are called formal. There are two classes of procedures:

- call procedure. The procedure is executed using a CALL-statement.
- function procedure. The procedure is executed using it as an operand in an expression-seven. The procedure calculates the value of the operand.

7.1 Procedure Attributes

There are two attributes which can only be used in the definition headings of procedures:

REENTRANT,
result-attribute.

The attribute REENTRANT indicates that several tasks can use the procedure at the same time. This implies that the procedure can be interrupted during its execution and called by a task of higher priority. After execution of the higher priority task execution for the lower priority task will be resumed where it was left off. Like the attribute RESIDENT, REENTRANT need not be written in specifications.

The result-attribute is obligatory for function procedures and must not occur anywhere else. It indicates the mode of the function result, which is then used as an operand in an expression-seven. It means that mode compatibility between the procedure calls and the procedure definition can be easily checked, both by the user and the compiler. The mode of the result must be simple-mode (see 4.3.2.1), which is the collective name for the modes of arithmetic, string and time data.

RESULT-ATTRIBUTE:
'RETURNS' ('SIMPLE-MODE').

7.2 Procedure Declaration

A procedure declaration defines one procedure. It consists of a declaration heading and a block-tail. The heading may comprise of a parameter list and attributes. The block-tail can contain block level declarations and statements, terminated by the key-word END.

FULL-PROCEDURE-DECLARATION:
LABEL-IDENTIFIER': 'PROCEDURE-DECLARATION';'.

PROCEDURE-DECLARATION:
'PROC' ['('((IDENTIFIER/'('IDENTIFIER/'(')')')'
 PARAMETER-MODE/'(')')')'
 ['RETURNS' ('SIMPLE-MODE')]
 ['GLOBAL' ['('GLOBAL-QUALIFIER')'] ['RESIDENT']]
 ['REENTRANT']'];
(LOCAL-IDENTIFIER-DECLARATION';')\$
STATEMENT \$
'END'.

Where:

LABEL-IDENTIFIER is the name of the procedure,

IDENTIFIER is the name of a formal parameter,

PARAMETER-MODE gives the mode attribute(s) of formal parameters
(see box below),

'RETURNS' ('SIMPLE-MODE') characterises a function (see 7.1),

GLOBAL-IDENTIFIER defines the library which will contain the procedure,

LOCAL-IDENTIFIER-DECLARATION is a declaration at block level.

PARAMETER-MODE:

['('INTEGER-CONSTANT//',';')']

((['INV'] (SIMPLE-MODE/

STRUCTURE-MODE) ['IDENTICAL'/'IDENT'])/

('DEVICE'/'DVC')DEVICE-TYPE)/

'FILE' [FILE-TYPE].

Where:

INTEGER-CONSTANT states an upper array boundary,

SIMPLE-MODE (see 4.3.2.1) is the collective name for the modes of
computational data,

STRUCTURE-MODE is described in 4.3.2.2.

According to parameter-mode a formal parameter can be

- an arithmetic, time or string entity,
- a structure,
- a device,
- a file,
- any kind of variable except semaphore,
- any kind of array except interrupt or signal.

The parameter mechanism used to describe the substitution of formal parameters for actual parameters is explained in 7.4.

The following procedure features are provided to increase program clarity:

1. A procedure has only one entry point.
2. The use of GOTO-statements to return from a procedure is not permitted.
3. A procedure declaration may contain several RETURN-statements (see 6.3./). This does not mean that there can be more than one return destination. The calling task or procedure is continued as if the procedure call was replaced by the procedure definition.

Differences between call procedures and function procedures are:

<i>Call procedure</i>	<i>Function procedure</i>
The CALL-statement is used to call the procedure.	The procedure call takes the form of an operand in an expression-seven.
No result-attribute is required in the heading of the procedure definition?	A result-attribute is obligatory in the heading of the procedure definition.
The RETURN-statement must not contain an expression-seven.	An expression-seven included in the RETURN-statement gives the result of the function procedure.
A RETURN-statement occurring immediately before END can be omitted.	END must not be reached during execution, i.e. at least one RETURN-statement is necessary.

Examples:

- 1) FPE: PROC(PAR(3)INV FIXED)RETURNS(FIXED)GLOBAL;
 DCL A FIXED;
 A = PAR(1)*PAR(2)*PAR(3);
 RETURN(A);
 END;
 T: TASK; DCL J(3)FIXED INIT(2,5,4), K FIXED INIT(5);
 K = FPE(J)-K;
- 2) A shorter version of the procedure declaration in (1)
 FPE: PROC(PAR(3)INV FIXED)RETURNS(FIXED)GLOBAL;
 RETURN(PAR(1)*PAR(2)*PAR(3));
 END;

7.3 Procedure Specification

A procedure specification defines one or more procedures which have been declared fully in another module. It consists solely of a heading, which enables the compiler to check the compatibility between procedure call and procedure specification.

```

('SPECIFY'/'SPC')((IDENTIFIER/'('IDENTIFIER/'','')')
PROCEDURE-MODE
['GLOBAL' ['GLOBAL-QUALIFIER')]]/'','');'.

```

```

PROCEDURE-MODE:
'ENTRY' ['('PARAMETER-MODE/'','')')
['RETURNS' ('SIMPLE-MODE')'].

```

Where:

IDENTIFIER is a procedure name

PARAMETER-MODE represents the attributes allowed for parameters
 (see last box in 7.2).

Example:

Specification of the procedure introduced as an example in the previous paragraph:

```

SPECIFY FPE ENTRY((3) INV FIXED)
RETURNS(FIXED) GLOBAL;

```

7.4 Procedure Call

As procedures are defined only at module level their identifiers are valid throughout the module. Therefore they may be entered from any task and can contain calls to other procedures.

A call procedure is called by creating a statement consisting of the key-word CALL (see 6.3.6) and the label-identifier of the procedure, followed by the actual parameters if they exist. See examples (1), (2).

A function procedure is called by using the label-identifier of the procedure (followed by the actual parameters if they exist) as an operand. See example (3).

Communication between a calling procedure or task and the called procedure can be achieved by:

- external data stored in files,
- data defined at module level,
- parameter transfer.

The following rules apply to the transfer of parameters:

1. In a procedure call all the actual parameters are assembled into a single bracketed list, separated by commas. On execution of a call, the first actual parameter in the list is associated with the first formal parameter in the list of the procedure definition, then the second actual parameter with the second formal parameter etc.
2. The numbers of actual and formal parameters must be the same.
3. If the actual parameter is an array the corresponding formal array must have the same number of subscripts and elements.

4. An actual parameter and its corresponding formal parameter must have the same mode, but may differ in other attributes.

Parameters can be classified according to their purpose:

- argument is a parameter used to transfer a value to the procedure from the calling task or procedure,
- result is a parameter used to return a value from the procedure to the calling task or procedure,
- transient is a parameter which acts as an argument but also transfers a result value from the procedure to the calling task or procedure.

The transfer of information between an actual and a formal parameter can be performed in two ways:

1. "call by value" (reference by value). The actual parameters provide values. The formal parameters are entities to which the values of the actual parameters are assigned when the procedure is entered.
The parameters can consist solely of arguments.
 - (a) The formal parameters are only permitted on the right side of an assignment, i.e. they are invariable for each procedure call. They are therefore given the attribute INV in the procedure heading.
 - (b) The formal parameters may occur on the left side of an assignment. Like variables they are initialised when the procedure is entered.
2. "call by reference" (reference by location). The actual parameters replace the formal parameters, which are merely symbols indicating where the actual parameters are to be inserted. That is the actual and formal parameters are equivalent.

If such actual parameters are defined at module level and the calling task is not of highest priority then during execution of the procedure the values of the formal parameters can be altered by a higher priority task interrupting the procedure.

- (a) Formal parameters are only allowed on the right side of an assignment. They are given the INV attribute and therefore can only be used as arguments.
- (b) Formal parameters may occur on the left side of an assignment. They are therefore either results or transients.

A parameter, which is called by reference and only supplies the procedure, should not be used as a variable inside the procedure (in order to store an intermediate result and thus save core by not using a local variable). This would change the parameter from an argument to a transient which could cause side effects if the actual parameter is used later outside the procedure without taking into consideration that it has been modified by the procedure.

The use of the attributes INV and IDENT (or IDENTICAL) in procedure headings is summarised in the following table. In the example the mode FIXED and the identifier X are given to a formal parameter.

	Formal parameter is	
	(a) invariable	(b) variable
1. call by value	X INV FIXED	X FIXED
2. call by reference	X INV FIXED IDENT	X FIXED IDENT

Examples:

Correct use of procedures:

```

1) CP1: PROC((PAR1,PAR2)INV FIXED,PAR3 FIXED IDENT):
    PAR3 = PAR1*PAR2*PAR3;
    END;
T: TASK;
    DCL K FIXED INIT(5)
    L FIXED INIT(4);
    CALL CP1(2,K,L);
    K=L-K;
    .
    .
    .

```

Parameter L is a transient.

The same result is obtained if the procedure heading is written as:

```
CP1: PROC(PAR1 INV FIXED, PAR2 INV FIXED IDENT,
          PAR3 FIXED IDENT);
```

In this latter case the second parameter is called by reference.

```
2) CP2: PROC((PAR1, PAR2) INV FIXED, PAR3 FIXED IDENT);
    PAR3 = PAR1 * PAR2;
    RETURN;
END;
T: TASK;
    DCL J FIXED INIT(2),
    K FIXED INIT(5),
    L FIXED;
    CALL CP2(J, K, L);
    K = L * 4 - K;
```

Parameter L is a result only.

```
3) FPE3: PROC(PAR1 FIXED, (PAR2, PAR3) INV FIXED)
    RETURNS(FIXED);
    PAR1 = PAR1 * PAR2 * PAR3;
    RETURN(PAR1);
END;
T: TASK; DCL K FIXED INIT(5),
    L FIXED INIT(4);
    K = FPE3(2, K, L) - K;
    .
    .
    .
```

All parameters are arguments only.

Incorrect use of procedures:

```
4) CP4: PROC((PAR1, PAR2) INV FIXED, PAR3 FIXED);
    PAR3 = PAR1 * PAR2 * PAR3;
END;
T: TASK; DCL K FIXED INIT(5),
    L FIXED INIT(4);
    CALL CP4(2, K, L);
```

Parameter PAR3 is only valid within the procedure, therefore the result is not accessible outside the procedure. Corrected in example (1).

```
5) CP5: PROC(PAR1 INV FIXED, PAR2 INV FIXED IDENTICAL,
          PAR3 FIXED IDENTICAL);
    PAR3 = PAR1 * PAR2;
END;
T: TASK; DCL K FIXED INIT(1),
    L FIXED;
    CALL CP5(2, 5 * K, L);
```

The second actual parameter delivers a value whereas the corresponding formal parameter requires a location.

```
6) T: TASK; DCL J(3) FIXED INIT(2, 5, 4);
    J(3) = FPE6(J) - J(2);
END;
```

```
FPE6: PROC(PAR(2) INV FIXED,
          PAR3 FIXED IDENT)
    RETURNS(FIXED);
    PAR3 = PAR(1) * PAR(2) * PAR(3);
    RETURN(PAR3);
END;
```

Number of actual parameters (one) must not be different from number of formal parameters (two).

7.5 Standard Functions

Standard functions are the procedures contained within the compiler or a program library and are accessible by PEARL programs. The list of standard functions is installation dependent. It is recommended that the following standard functions should always be included:

'ABS' ('EXPRESSION-SEVEN')	calculates the absolute value of the expression-seven
'SIGN' ('EXPRESSION-SEVEN')	obtains the sign of the expression-seven +1 for <i>expr.-seven</i> > 0 0 for <i>expr.-seven</i> = 0 -1 for <i>expr.-seven</i> < 0
'SQRT' ('EXPRESSION-SEVEN')	calculates the square root of the expression-seven
'SIN' ('EXPRESSION-SEVEN') 'COS' ('EXPRESSION-SEVEN')	calculates the sine or cosine of the expression-seven
'ARCTAN' ('EXPRESSION-SEVEN')	
'LN' ('EXPRESSION-SEVEN')	calculates the natural logarithm of the expression-seven
'EXP' ('EXPRESSION-SEVEN')	calculates the exponential function of the expression-seven.

These functions operate on arguments of mode *FIXED* as well as *FLOAT*. The resulting values will be of mode *FLOAT*, except for *SIGN* which yields a *FIXED* value. As these functions are defined externally they may be used without explicit declarations and specifications.

If an identifier already occupied by a standard function is used to denote another program entity the standard function becomes invalid in the area where this entity is defined. Because standard functions are assumed to be global at module level and an identifier can only denote one entity in one area, identifiers of standard functions must not be used for other entities at module level, only at nested levels.

PART III: INPUT/OUTPUT FACILITIES

8. THE SYSTEM DIVISION

The system division of PEARL provides the description of the hardware configuration required in a specific program. That is the standard peripherals, the process peripherals, the sensors, effectors, interrupts and signals. The device connections are specified by system specific names.

In the majority of cases, the programmer is not so much interested in the detailed wiring of the total hardware configuration, as in the connections of specific devices which provide his problem specific data. PEARL enables the programmer to declare problem specific designators for devices and device connections which can then readily be used in the problem division of his PEARL program. Thus a meaningful declaration of the process dependent data is guaranteed, which in return, significantly enhances the legibility of the PEARL program.

By describing the hardware configuration within the system division of a PEARL program another advantage becomes apparent as soon as changes in the hardware configuration have to be made. That is that changes only have to be made to the system division. Furthermore the system division of a PEARL program provides for the automatic generation of a problem specific executive system.

8.1 Configuration Description in the System Division

In PEARL the hardware configuration of an automation system is described in the system division by defining the various connections and data routes between devices or device groups. Because of the clearness and generality of the underlying concept this can be achieved very easily and provides a powerful programming tool.

Irrespective of any peculiar characteristic pertaining to a particular device it will be assumed that any device exchanges data with other devices through SINGLE-CONNECTIONS at the device ports, a specific data flow direction being associated with each of these connections. A similar assumption is made for arrays and groups of devices, which leads to the logical construction of ARRAY-CONNECTIONS and GROUP-CONNECTIONS.

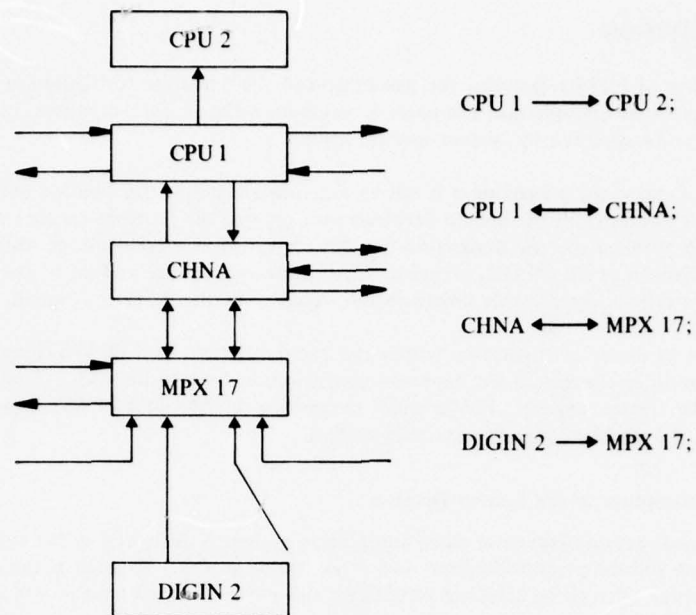
Syntax:

CONNECTION:	
SINGLE-CONNECTION	TRANSFER-DIRECTION [SINGLE-CONNECTION] / TRANSFER-DIRECTION
ARRAY-CONNECTION	SINGLE-CONNECTION / TRANSFER-DIRECTION
[GROUP-CONNECTION]	[GROUP-CONNECTION] / TRANSFER-DIRECTION
	ARRAY-CONNECTION ';'. .
TRANSFER-DIRECTION:	
	'←' /
	'→' /
	'↔'.

Functional Explanation:

- SINGLE-CONNECTION specifies a device of the configuration
- ARRAY-CONNECTION and GROUP-CONNECTION specify a device array and a device group resp.
- TRANSFER-DIRECTION specifies the direction of information flow.

Examples of SINGLE-CONNECTIONS for the illustrated hardware configuration.

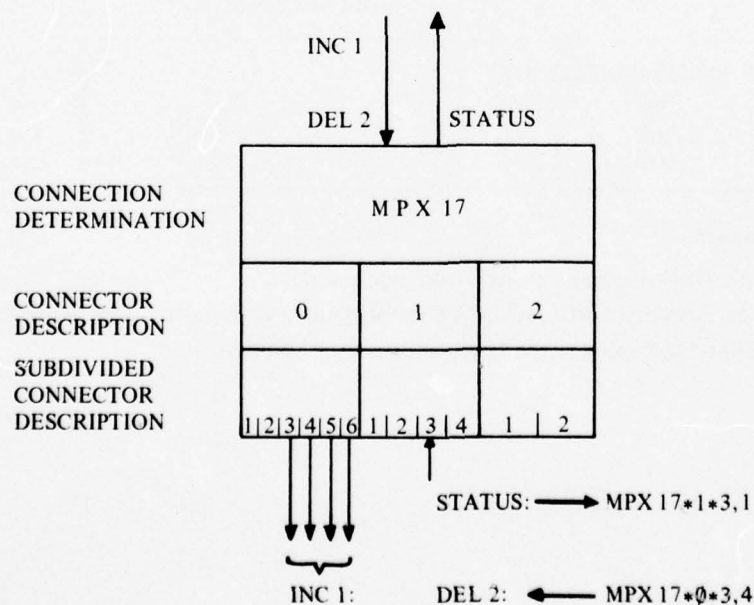


Based on this very general concept PEARL offers easy-to-handle facilities for a more detailed and problem-oriented connection and routing description of both single-devices and for device arrays/groups.

Information, as it is transferred back and forth (as described above) between single devices can be given logical problem-oriented names.

Example: STISTATUS: DIGIN 2 \longrightarrow MPX 17

The connection and routing itself can be described in more detail, but will still be device independent. In this PEARL subset it is assumed that any SINGLE-CONNECTION is subdivided into two further levels, the connector level and the sub-connector level. This provides a quick and easy method of specifying data at the required level of detail, be it word, byte, bit, channel, subchannel or single line level.



8.2 Connection of Single Devices

The syntax for specifying single devices, together with examples, is summarized in the following table.

Syntax for single devices:

Examples:

SINGLE-CONNECTION: CONNECTION-IDENTIFICATION \$ + CONNECTION-DETERMINATION + CONNECTOR-DESCRIPTION.	SENS1:SENS2:SENSOR: ANIN(3)*CH13*12,2
CONNECTION-IDENTIFICATION: IDENTIFIER ':'	SENS1:
CONNECTION-DETERMINATION: CONNECTION-TYPE ['(' INTEGER-CONSTANT ')'] .	ANIN(3)
CONNECTION-TYPE: IDENTIFIER.	ANIN
CONNECTOR-DESCRIPTION: '*' (INTEGER-CONSTANT/IDENTIFIER) + SUBDIVIDED-CONNECTOR-DESCRIPTOR.	*CH13*12,2
SUBDIVIDED-CONNECTOR-DESCRIPTOR: '*' INTEGER-CONSTANT '.' INTEGER-CONSTANT.	*12,2

Functional Explanation:

- The CONNECTION-IDENTIFICATION contains designators for devices, which can then be used in the problem division. Different names may be used for the same device.
- The CONNECTION-DETERMINATION provides a distinction between devices on the basis of how they are connected to each other. These connection types are known to the compiler and are fully described in the system manual of the particular installation. The CONNECTION TYPE contains all the information necessary to drive the device.
- If, in one system, several devices share the same connection type, numbers are used to differentiate between them. The selection is then done using these numbers, which are enclosed in brackets.
- The CONNECTOR-DESCRIPTION provides a more detailed device description than that given in the CONNECTION-TYPE specification. Following the first asterisk either an input or output connection for a device, or several input or output connections for a subdevice, can be specified. Numbers following the second star describe the input and output connections for a subdevice. In this case, the number before the comma specifies the first position of the input and output connections of a subdevice, and the second specifies the number of connections involved for this subdevice. Thus a single device is identified by
 - device type * device input/output connections, or
 - device type * subdevice * subdevice input/output connections.

8.3 Connection of Device-Arrays/Groups

Syntax for device groups:

ARRAY-CONNECTION: IDENTIFIER ARRAY-BOUNDS ':' + CONNECTION-TYPE ARRAY-BOUNDS.
ARRAY-BOUNDS: '(' INTEGER-CONSTANT '.' INTEGER-CONSTANT ')'. cont'd/.....

...../cont'd

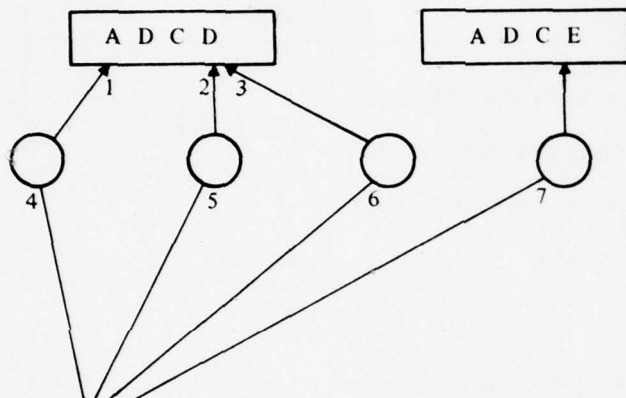
GROUP-CONNECTION:
 CONNECTION-DETERMINATION
 ['*' (INTEGER-CONSTANT/ARRAY-BOUNDS)
 {SUBDIVIDED-CONNECTOR-DESCRIPTOR}] //'+'.

Functional Explanation:

- ARRAY-BOUNDS specifies the bounds of a device array. The first number specifies the first element of the device array and the second specifies the last element. The index goes up in steps of one.
- The identifier with ARRAY-BOUNDS at ARRAY-CONNECTION contains a designator for a device array which can be used by the programmer in the problem division.
- A device group can be specified as either, device types and device I/O connections of the same device type (see example 1), or as subdevices and I/O connections of subdevices of the same type (see example 2).

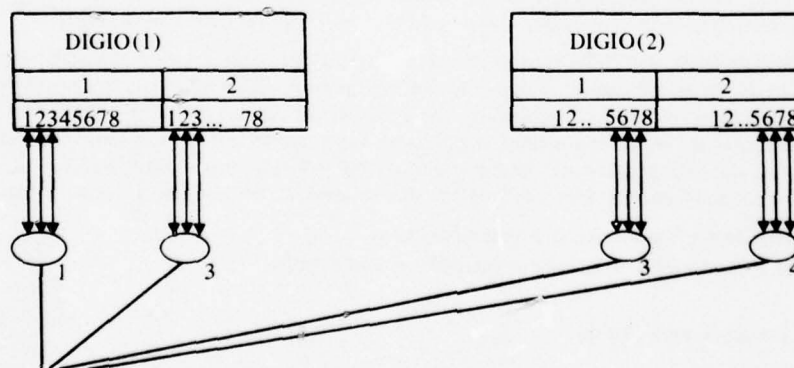
Examples:

1)



ANALOGTERMINAL(4:7): \rightarrow ADCD*1+ADCD*2+ADCD*3+ADCE;
 ANALOGTERMINAL(4:7): \rightarrow ADCD*(1:3)+ADCE;

2)



DIGITALTERMINAL(1:4): \leftrightarrow DIGIO(1)*1*1,3
 +DIGIO(1)*2*1,3
 +DIGIO(2)*1*5,3
 +DIGIO(2)*2*5,3;

DIGITALTERMINAL(1:4): \leftrightarrow DIGIO(1)*(1:2)*1,3+DIGIO(2)*(1:2)*5,3;

9. DEVICES

The device identifiers specified in the system division can be used in the problem division. The identifiers describe interrupts, signals or devices and, together with their associated attributes, have to be declared in the problem division.

9.1 Device Attributes

A device can have the following attributes assigned to it:

Syntax:

DEVICE-TYPE: ['INTERNAL'] ('SOURCE'/'SINK'/'SOUSI') ['DIRECT'] [SIMPLE-MODE].

Functional Explanation:

- INTERNAL indicates that it is unimportant in which form the data is represented and that it is only necessary that the form of representation is compatible to the data types from the simple mode.
- The attributes SOURCE, SINK and SOUSI provide a description of the flow of data to or from the device. SOURCE indicates that the device is a data source, SINK that it is a data sink, and SOUSI that it is both source and sink.
- The mode of access to the device is specified using DIRECT. DIRECT allows random access of information from the device.
- SIMPLE-MODE indicates the type of data and how it is to be represented for use in the program.

9.2 Declaration of Devices

In the system division a distinction is made between system specific device designators and problem specific device designators. Whilst the system specific device designators are known (and used) only in the system division, the problem specific device designators can also be used in the problem division. All devices must have been declared before their problem specific device designators can be used.

Syntax for Device Declarations in the Problem Division:

DEVICE-DECLARATION: ('DECLARE'/'DCL') (ONE-IDENTIFIER-OR-LIST DEVICE-OR-INTERRUPT-OR-SIGNAL-MODE [GLOBAL-ATTRIBUTE] //'') ':'.
--

DEVICE-OR-INTERRUPT-OR-SIGNAL-MODE: [INTEGER-IN-BRACKETS] ('INTERRUPT'/'IRUPT' / 'SIGNAL' / DEVICE-MODE.

DEVICE-MODE: ('DEVICE'/'DVC') DEVICE-TYPE.

Examples:

```

DCL (SWITCH, LAMP) DEVICE SINK BIT(1) GLOBAL;
DCL VALVE DEVICE SINK BIT(2);
DCL IR INTERRUPT GLOBAL;
DCL ENDOFFILE SIGNAL;
DCL DETECTOR DVC SOURCE FIXED GLOBAL;
  
```

Functional Explanation:

- ONE-IDENTIFIER-OR-LIST contains problem specific device identifiers from the system division.
- INTEGER-IN-BRACKETS is the device array subscript.
- In the system division all identifiers are interpreted as device identifiers but in the problem division a distinction is made between interrupts, signals and devices. Interrupts and signals have to be distinguished because of their different effects on tasks and their effect when the ON prefix is used. Devices demonstrate some similarity to files.

9.3 Specification of Devices in the Problem Division

The problem specific device identifiers must be specified as global when they have been declared in another module.

Syntax for Device Specifications in the Problem Division:

DEVICE-SPECIFICATION: ('SPECIFY'/'SPC') (ONE-IDENTIFIER-OR-LIST DEVICE-OR-INTERRUPT-OR-SIGNAL-MODE [GLOBAL-ATTRIBUTE] //'').
DEVICE-OR-INTERRUPT-OR-SIGNAL-MODE; [INTEGER-IN-BRACKETS] ('INTERRUPT'/'IRUPT'/ 'SIGNAL'/ DEVICE-MODE).
DEVICE-MODE: ('DEVICE'/'DVC') DEVICE-TYPE.

Examples:

```
SPECIFY (SWITCH, LAMP) DEVICE SINK BIT(1) GLOBAL (XY);
SPC IR INTERRUPT GLOBAL (XY);
```

Functional Explanation:

- The semantics are very similar to that of the device declaration.
- The GLOBAL attribute contains the module within which the problem-specific device identifier has been declared.

10. FILES (DATA STATIONS)

10.1 Definition of Files (Distinction between Devices and Files)

Input/output operations transfer data between data stations. A data station consists of a data container and an access mechanism. Access of the data from the data container is only performed via this mechanism. When information is transferred to a device the physical information flows into the data container of the device. This means the information is transferred to the device without an intermediate station. Management of the devices is controlled by the user.

During the transfer of information to a file the user does not know which physical address the information is delivered under. Thus, the user does not have to take care of device organization. This is done for him by file management routines. He defines a file with data container and access mechanism, and works on this level. He does not have to organize data storage on the physical device.

The data container of a file is subdivided into blocks which can be accessed using the I/O operations. The length and organization of these blocks is not standardized.

10.2 File Attributes

File attributes must be specified to define the data in the data containers and to define the access mechanisms.

Syntax:

FILE-TYPE: FILE-CLASS FILE-USAGE FILE-ACCESS [FILE-CONTROL] [FILE-CONTINUATION] {FILE-DIMENSION} FILE-CONTENTS.
FILE-CLASS: 'ANALOGUE'/'EXTERNAL'/'INTERNAL'.
FILE-USAGE: 'SOURCE'/'SINK'/'SOUSI'.
FILE-ACCESS: 'DIRECT'/'FORWARD'.
FILE-CONTROL: SIGNAL '(' (SIGNAL-IDENTIFIER/'(',') ') '.
FILE-CONTINUATION: ('STREAM'/'NOSTREAM') 'NOCYCL' .
FILE-DIMENSION: '(' (INTEGER-CONSTANT/'(',') ') '.
FILE-CONTENTS: LOCAL-MODE.

Functional Explanation:

- **FILE-CLASS** specifies how the data is represented in the data container, as follows:
 - ANALOGUE** analogue representation (e.g. for graphic applications)
 - EXTERNAL** discrete representation (e.g. of print codes)
 - INTERNAL** in this case the form of representation is irrelevant as long as it is compatible with the data types being used
- **FILE-USAGE** describes the ways in which a file can be accessed:
 - SOURCE** allows only reading
 - SINK** allows only writing
 - SOUSI** allows both reading and writing
- **FILE-ACCESS** describes the mode of accessing the blocks of a file:
 - DIRECT** random access of blocks in the file
 - FORWARD** only the next block of the file can be accessed (serial)
- **FILE-CONTROL** controls the access of data from a file. If an illegal access is attempted it will be reported via one of the specified **SIGNAL-IDENTIFIERS**. **END-OF-FILE** will also be reported in this way.
- **FILE-CONTINUATION** controls the automatic index switching between the blocks of a file, as follows:
 - STREAM** automatic switching to the next block
 - NOSTREAM** suppression of automatic switching
 - NOCYCL** suppression of cyclic switching to first block of file at end-of-file.
- **FILE-DIMENSION** is used to define the number of blocks, the number of lines per block, and the number of data elements per line.
- **FILE-CONTENTS** is used to specify the data type or data structure of the data in the data container.

10.3 Declaration and Specification of Files

Files are declared and specified as follows:

Syntax:

```
FILE-DECLARATION:
    ('DECLARATION' / 'DCL')
    (ONE-IDENTIFIER-OR-LIST
     'FILE' [FILE-TYPE]
     [GLOBAL-RESIDENT-ATTRIBUTE]
     // ';' '.').
```

Examples:

```
DCL TAPE1 FILE INTERNAL SOUSI FORWARD CHAR(6) GLOBAL;
DECLARE FIXY FILE;
DCL DISPLAY FILE ANALOGUE SOUSI DIRECT BIT(8) GLOBAL;
DCL F4 FILE GLOBAL;
```

Functional Explanation:

- ONE-IDENTIFIER-OR-LIST is a list of file names.

In order to introduce to a module a file which has already been declared in another module, the file specification must be used.

Syntax:

```
FILE-SPECIFICATION:
    ('SPECIFY' / 'SPC')
    (ONE-IDENTIFIER-OR-LIST
     'FILE' [FILE-TYPE]
     [GLOBAL-ATTRIBUTE]// ';' '.').
```

Examples:

```
SPC(FIXY, F4)FILE GLOBAL (XY);
SPECIFY TAPE1 FILE GLOBAL (XY);
```

Functional Explanation:

- ONE-IDENTIFIER-OR-LIST represents the file name.

10.4 Creation and Deletion of Files

The creation of files on storage media can be achieved by using the CREATE statement.

Syntax:

<pre>CREATE-STATEMENT: 'CREATE' (TITLE / FILE-IDENTIFIER) UPON.</pre>
<pre>TITLE: 'TITLE' '(' CHARACTER-STRING-CONSTANT ')' [FILE-TYPE].</pre>
<pre>UPON: 'UPON' DEVICE-IDENTIFIER.</pre>

Examples:

```
CREATE TAPE1 UPON TAPE;
CREATE TITLE ('MESSAGE-FILE') UPON DISK;
```

Functional Explanation:

- DEVICE-IDENTIFIER denotes the medium on which the required storage area is to be reserved as specified in the file type specification (see para. 10.2).
- TITLE indicates that a file is to remain reserved until explicitly cancelled later in the program. FILE-IDENTIFIER indicates that a file is to remain reserved until the end of the program.
- The CREATE statement is only valid for a file which does not already exist. The deletion of files on storage media is done via the DELETE statement.

Syntax:

DELETE-STATEMENT: 'DELETE' (TITLE/FILE-IDENTIFIER).

Examples:

```
DELETE TAPE1;
DELETE TITLE ('MESSAGE-FILE');
```

Functional Explanation:

- The FILE as specified by TITLE/FILE-IDENTIFIER is deleted and the subsequent creation of a new file with the same name is now legal.
- Deletion of an open file implicitly means automatic closing of this file.

10.5 Opening and Closing of Files

Access to a file is only valid after it has been opened using the OPEN statement. An opened file must be closed using the CLOSE statement. All I/O operations on closed files are illegal.

Syntax:

OPEN-STATEMENT: 'OPEN' FILE-IDENTIFIER [TITLE UPON].
CLOSE-STATEMENT: 'CLOSE' FILE-IDENTIFIER.

Examples:

```
OPEN TAPE1;
OPEN MESSAGES TITLE ('MESSAGE-FILE') UPON DISK;
CLOSE TAPE1;
CLOSE MESSAGES;
```

Functional Explanation:

- The OPEN and CLOSE statements respectively open or close the file specified in FILE-IDENTIFIER.
- To access a file the FILE-IDENTIFIER is sufficient to uniquely identify it.
- When opening a file that has been created with TITLE then the open statement must also include this TITLE specification. The file attributes must be known whenever the file is to be opened.

11. INPUT/OUTPUT OF PROCESS SIGNALS

11.1 Statements for Direct Process Input/Output

The transfer of data between devices and objects is performed using the TAKE and the SEND statements.

Syntax:

DIRECT-TAKE-STATEMENT:
'TAKE' 'FROM' DEVICE-ELEMENT ['TO' SYMBOL].

Example: TAKE FROM DETECTOR TO K;

Functional Explanation:

- The DIRECT-TAKE-STATEMENT causes the transfer of data from the device to the specified object in the PEARL program.
- DEVICE-ELEMENT specifies the device.
- SYMBOL specifies the object in the PEARL program. Permissible objects are variables, array elements, structure elements and arrays.
- If the SYMBOL specification is missing, the TAKE statement transfers the information into a storage cell or array assigned to the device.

Syntax:

DIRECT-SEND-STATEMENT:
'SEND' ['FROM' SYMBOL]
'TO' DEVICE-ELEMENT.

Examples:

SEND FROM BIT1 to SWITCH;
SEND FROM BIT2 TO VALVE;

Functional Explanation:

- The DIRECT-SEND-STATEMENT causes the transfer of data from objects in the PEARL program to specified devices.
- SYMBOL specifies the object in the PEARL program. Permissible objects are variables, array elements, structure elements or arrays.

If the SYMBOL specification is missing, the SEND statement initiates the transfer from the storage cell or array assigned to the device.

11.2 Statements for Organized Process Input/Output

Because of the hardware and physical limitations of the various devices it is usually necessary to transform and convert the data before a transfer can take place. For instance A/D converters are limited to a specific range of numbers. These data transformations can be performed immediately after the data transfer using the I/O statements of PEARL. This process of input/output together with data conversion and adaptation is known as organized process input/output.

Syntax:

TAKE-STATEMENT:
DIRECT-TAKE-STATEMENT ['THRU' P-OPTION].

P-OPTION:
FORMAT-PROCEDURE-IDENTIFIER
['(' EXPRESSION-SEVEN-PACK ')'].

SEND-STATEMENT:
DIRECT-SEND-STATEMENT ['THRU' P-OPTION].

Example: TAKE FROM DETECTOR TO K THRU GAUGE (NORM);

Functional Explanation:

- FORMAT-PROCEDURE-IDENTIFIER specifies the procedure and parameters required for the adaptation of the device data.

12. STANDARD INPUT/OUTPUT

12.1 Composition of Formatted I/O Statements

The GET and PUT statements are used for reading from and writing to opened files. The GET statement controls input to the computer, the PUT statement output from the computer. In formatted I/O, the transformation between internal and external data representation is managed using formats (see para. 12.2).

Syntax:

<p>GET-STATEMENT: 'GET' 'FROM' FILE-ELEMENT 'TO' ONE-SYMBOL-OR-LIST ['THRU' STANDARD-FORMAT-LIST].</p>
<p>FILE-ELEMENT: FILE-IDENTIFIER ['AT' '(' FILE-INDEX //',' ') '].</p>
<p>FILE-INDEX: ['HERE' ('+' / '-')] INTEGER-EXPRESSION-SEVEN.</p>
<p>PUT-STATEMENT: 'PUT' 'FROM' ONE-SYMBOL-OR-LIST 'TO' FILE-ELEMENT ['THRU' STANDARD-FORMAT-LIST].</p>
<p>ONE-SYMBOL-OR-LIST: SYMBOL / '(' (SYMBOL//',' ') '.</p>

Functional Explanation:

- ONE-SYMBOL-OR-LIST indicates the objects to be transferred to (or from) the file from (or to) the location specified in FILE-ELEMENT. For the following, the objects in ONE-SYMBOL-OR-LIST will be referred to as a data list.
- STANDARD-FORMAT-LIST describes the format of the data in the data list. For each element in the data list one format specification is required in the format list. The formats are described in para. 12.2.
- FILE-ELEMENT specifies the position of the data (by file name) and eventually a position specification.
- FILE-INDEX indicates the page, line and element position. Each component of this specification can be positioned in two ways:
 - 1) absolute: INTEGER-EXPRESSION-SEVEN specifies the absolute position in the file,
 - 2) relative: INTEGER-EXPRESSION-SEVEN specifies a position in relation to the last position. The plus and minus signs specify forward and backward positioning.

Examples:

GET FROM TAPE1 AT (43,5,1) TO (WORDS, TEXTS) THRU (2)A(6);
PUT FROM (WORDS, TEXTS) TO TAPE1 THRU (A(6),A(6));

12.2 Formats

Formats are used to define the transformation required between internal and external data representations.

- A Character string format (character set: STRING-CHARACTER),
- B Bit string format (character set: BINARY-DIGIT),
- B3 Octal string format (character set: OCTAL-DIGIT),
- B4 Sedecimal string format (character set: SEDECIMAL-DIGIT).

— Number formats:

The first integer within the brackets specifies the field length of the decimal number in the external representation including the sign position. The second integer specifies the number of positions after the decimal point in the external representation.

- F Fixed point number
Describes the external representation of a FLOAT or FIXED number with sign.
- E Floating point number
Describes the external representation of a floating point number. The internal representation is specified using a declaration with the attribute FLOAT. The exponent is an integer number with *n* digits, where *n* is a machine-dependent constant.

— Time formats:

- D Duration
Describes the external representation of a duration. The internal representation is implementation dependent and the units are either seconds or tenths of seconds.

The first integer within the brackets specifies the total field length. The second integer specifies the number of positions following the decimal point for the specification, in units of tenths of seconds. The hour, minute and second representations must each be two digits.

Example: Printing in format D (24,1):
 'PUT' TIMEDURATION 'TO' SD 'THRU' (D(24,1));
Result is: ...9 HRS 12 MIN 46.3 SEC.
- T Time
Describes the external representation of a time. The integer in brackets is the total field size.

Example: Printing in format T (10):
 'PUT' TIME 'TO' SD 'THRU' (T(10));
Result is: ...9:46:46.

13. GRAPHIC INPUT/OUTPUT

13.1 Composition of Graphic I/O Statements

Monitoring of processes is being done more and more often using graphic representations. The main reason for this is that the man/machine communication is much better in graphic form than in long strings of digits. Thus in many applications today graphic I/O is desirable. In most cases the graphic devices only permit graphic output, for instance a plotter or a growing table. Graphic input is rarely used since it introduces a lot of difficult identification problems. But graphic input permits the dialogue between man and computer to be on a graphic basis and is being applied more and more often for computer aided design. Graphic input can be done by using a light pen, graphic display or digitiser. All three methods mean inputting a continuous stream of points or vectors.

PEARL provides facilities for both graphic output and graphic input.

Syntax:

DRAW-STATEMENT: 'DRAW' 'FROM' ONE-SYMBOL-OR-LIST 'TO' FILE-ELEMENT 'THRU' GRAPHIC-FORMAT-LIST.
SEE-STATEMENT: 'SEE' 'FROM' FILE-ELEMENT 'TO' ONE-SYMBOL-OR-LIST 'THRU' GRAPHIC-FORMAT-LIST.

Functional Explanation:

- ONE-SYMBOL-OR-LIST consists of a list of variables, field elements or fields which represent either point coordinates or levels of brightness for the graphic device.
- FILE-ELEMENT describes a file and the position in that file in which the data of the ONE-SYMBOL-OR-LIST specification (at the DRAW-STATEMENT) is to be stored, or
FILE-ELEMENT describes the display file, i.e. the file and the location assigned to the graphic device, from which the information for the variable field elements or fields of the ONE-SYMBOL-OR-LIST are to be taken (SEE-STATEMENT).
- DRAW-STATEMENT indicates the graphic output of data. SEE-STATEMENT indicates the graphic input of user data into the computer.
- GRAPHIC-FORMAT-LIST see para. 13.2.

13.2 Graphic Formats

The graphic field elements define

- (a) the semantics of the data in the data list of the DRAW or SEE statement,
- (b) the control of the input/output to or from the graphic device.

Syntax:

GRAPHIC-FORMAT-LIST: [INTEGER-IN-BRACKETS] (G-FORMAT-ELEMENT/ G-FORMAT-LIST).
G-FORMAT-LIST: '(' GRAPHIC-FORMAT-LIST // ';') '.
G-FORMAT-ELEMENT: 'FRAME' / 'BR' / 'MAP' / 'XA' / 'XR' / 'XS' / 'XI' / 'XP' / 'YA' / 'YR' / 'YS' / 'YI'.

Functional Explanation:

- INTEGER-IN-BRACKETS is the repetition factor for the subsequent G-FORMAT-ELEMENT or the subsequent G-FORMAT-LIST.
- Formats for picture control
FRAME cancels the picture on the device. All information up to the next FRAME format will be drawn in the same picture. MAP controls the transfer of the graphic information to the device or the transfer of information from the device to the computer.
- Formats for the positioning of the drawing pen
XA, YA The assigned data elements in the data list specify the coordinates of a point in absolute picture units.
XR, YR The assigned data elements in the data list specify the coordinates of a point in relative picture units (in relation to the last point).
XI, YI The X and Y coordinates of the pen are incremented at the generation of each new point. The increments are as specified in the data list.
XP Positions the drawing pen at the start of the plotting axes.
- Formats for the scaling of units
XS, YS The coordinates of the points are multiplied by the factors as specified in the data list. The scaling of units ensures that the graphic representation fits on the display and vice versa.
- Control of brightness
BR On some graphic devices a point can be displayed at various levels of brightness. The related data element specifies the level of brightness required, until it is changed by the program.

PART IV: PARALLEL PROGRAMMING FEATURES

14. TASKS

14.1 Introduction of Tasks

As already explained in para. 2.4 program systems for monitoring and controlling processes and experiments require more than just a sequential execution of a program: They have to be organized as a set of distinct program parts to be executed autonomously, but requiring coordination with other program parts from time to time. Amongst these program parts parallel (or semi-parallel) execution is inevitably needed in order to make best use of the system's capabilities and to guarantee the system's optimal performance.

Within PEARL, program parts of this kind are called "tasks". Tasks are available for the PEARL programmes by using the set of facilities for coordination and synchronization, described in detail in Chapter 17. The tasks of a program system are executed under the control and management of the operating system. Since these tasks are largely independent from each other they compete for the system's resources. Precedence amongst tasks is decided by the operating system, according to their respective priorities.

Management and execution of tasks is based upon a model scheme which is explained in para. 2.4.1.

14.2 Declaration of Tasks

Declaration of a task involves the assignment of a name to a series of PEARL statements (the task body). The execution of the task body represents a task.

A task can only be declared at the module level. That is, there can be no task declarations within a procedure declaration, or within a task declaration and hence, subtasks are not permitted.

Syntax:

FULL-TASK-DECLARATION: LABEL-IDENTIFIER: TASK-DECLARATION.
TASK-DECLARATION: 'TASK' [PRIORITY] [GLOBAL-RESIDENT-ATTRIBUTE] ';' BLOCK-TAIL ';'
PRIORITY: ('PRIORITY' / 'PRIO') INTEGER-CONSTANT.

Example: Declaration of a task called "ALARM" with priority 5.

```
ALARM: TASK PRIO 5 GLOBAL RESIDENT;
/*SWITCH-ON LAMP*/
DCL BIT1 BIT(1) INIT ('1'B);
SEND FROM BIT1 TO LAMP;
END;
```

Functional Explanation:

- The IDENTIFIER represents the name of the task.
- The task priority is specified as a positive integer, following the key-words PRIORITY or PRIO. The smaller the value the higher the priority.
- The GLOBAL attribute indicates that the task can be used in external modules.
- The RESIDENT attribute indicates that the task resides in core permanently, even when not activated.
- The BLOCK-TAIL contains task specific declarations and statements. It must contain at least one statement. A task will be terminated automatically, when the end of its block-tail has been reached.

14.3 Specification of Tasks

A task is known primarily within the module in which it has been declared. If it has been declared to be GLOBAL then it may also be used in external modules, but only if the task has been specified within the external module by means of a task specification.

Syntax:

```

TASK-SPECIFICATION:
('SPECIFY'/'SPC')
(ONE-IDENTIFIER-OR-LIST 'TASK'
['GLOBAL' ['(' IDENTIFIER ')']] //',' ') ';

```

Example: Specification of the global task ALARM from para. 14.2 which is assumed to be declared in module MODUL3, an external module.

SPECIFY ALARM TASK GLOBAL (MODUL3);

Functional Explanation:

- ONE-IDENTIFIER-OR-LIST identifies a single task name or a list of task names.
- The GLOBAL-ATTRIBUTE indicates in which external module the specified task has been declared.
- After the specification all task operations may be performed on the task within the external module.

15. SCHEDULING OF TASKS

One of the main assets of PEARL, when compared to any conventional programming language is its comprehensive facilities for handling time- and event-dependent conditions. Through its intrinsic scheduling mechanisms, PEARL offers a powerful set of facilities for the control and management of tasks.

15.1 Time-Dependent Scheduling

PEARL scheduling features permit the order of program execution to be changed at

- points of time (time of day, clock-time),
- after specified durations of time, or
- on the occurrence of interrupts.

Syntax:

```

SCHEDULE-1:
  SCHEDULE-2
  +
  REPETITION-OPTION.

SCHEDULE-2:
  'AT' CLOCK-EXPRESSION-SEVEN /
  'AFTER' DURATION-EXPRESSION-SEVEN /
  'WHEN' IDENTIFIER [INTEGER-IN-BRACKETS].

REPETITION-OPTION:
  'ALL' DURATION-EXPRESSION-SEVEN
  ['UNTIL' CLOCK-EXPRESSION-SEVEN /
  'DURING' DURATION-EXPRESSION-SEVEN].

```

Examples:

```

AT      7:3:50
AFTER   8 SEC
WHEN    IR
ALL 5 MIN UNTIL 7:3:50

```

Functional Explanation:

- The AT condition takes effect at the point of time specified in the CLOCK-EXPRESSION-SEVEN.
- The AFTER condition takes effect after a time interval of DURATION-EXPRESSION-SEVEN time units. The time interval starts with the evaluation of the AFTER condition.

- The IDENTIFIER with index specification in the WHEN condition specifies an interrupt (see para. 9.2).
- The ALL condition effects execution immediately. It takes effect again at intervals of time specified in the first DURATION-EXPRESSION-SEVEN specification, UNTIL the time specified in CLOCK-EXPRESSION-SEVEN has been reached, or DURING the time interval specified in the DURATION-EXPRESSION-SEVEN. Using the ALL condition, programming of cyclic tasks is simplified.

15.2 Event-Dependent Scheduling

In the event-dependent scheduling of tasks in PEARL, an event can be an interrupt or a signal (as defined in the system division). The effect of interrupts is different to that of signals. An interrupt may cause one or more dormant tasks, as specified in the WHEN condition, to be activated, whereas a signal causes a sequence of statements within a task or procedure body which is assigned to the signal, to be executed.

Lock-out and resume facilities are provided to permit the manipulation of interrupts. The programmer can also stimulate interrupts and signals using these two facilities.

Syntax:

WHEN-CONDITION: 'WHEN' IDENTIFIER [INTEGER-IN-BRACKETS].
ON-PREFIX: ('ON' SIGNAL-IDENTIFIER [INTEGER-IN-BRACKETS] ':') \$.
INTERRUPTION: ('ENABLE' / 'DISABLE' / 'TRIGGER' / 'INDUCE') SYMBOL.

Examples:

```

WHEN IR ACTIVATE ALARM;
ON ENDOFFILE: GOTO FILELABEL;
ENABLE IR;
DISABLE IR;
TRIGGER IR;
INDUCE ENDOFFILE;

```

Functional Explanation:

- The IDENTIFIER assigned in the WHEN condition represents an interrupt.
- Several tasks may be specified in the same WHEN condition. When an interrupt occurs all the tasks waiting for the interrupt will be activated or continued.
- The ON prefix is only permitted at the highest task and procedure levels.
- On exit from a procedure or on termination of a task the ON prefix becomes ineffective.
- SYMBOL specifies an interrupt in the ENABLE, DISABLE and TRIGGER statements and a signal in the INDUCE statement.
- The DISABLE statement prevents the specified interrupts from fulfilling any WHEN condition related to it.
- The ENABLE statement overrides a previous DISABLE statement.
- By means of a TRIGGER statement the programmer can induce interrupts, e.g. for fulfilling the tasks' WHEN conditions (simulated interrupts).
- Using the INDUCE statement the programmer can induce signals, thus forcing a branch to the associated ON prefix.

16. TASK CONTROL STATEMENTS

Management and execution of tasks within the model scheme of task states (see para. 2.4.1) can be controlled by the programmer by using the following PEARL statements:

ACTIVATE	
SUSPEND	RESUME
CONTINUE	PREVENT
TERMINATE.	

These task control statements provide the programmer with a powerful means of managing his tasking requirements.

16.1 Activation of Tasks

Starting a task in PEARL is done using the ACTIVATE statement. It transforms a task from the "dormant" to the "runnable" state (see para. 2.4.1).

Through the time-dependent scheduling mechanism, as described in para. 15.1, PEARL offers a very comprehensive set of time, duration, event and repetition conditions, using which the programmer can satisfy his task activation requirements.

Syntax:

ACTIVATE-STATEMENT:
[SCHEDULE-1] 'ACTIVATE' IDENTIFIER [PRIORITY].

Note: For details on SCHEDULE-1 see para. 15.

Examples:

	ACTIVATE ALARM PRIO 6; /*UNCONDITIONED*/
AT 7:3:50	ACTIVATE ALARM ; /*TIME*/
AFTER 8 SEC	ACTIVATE ALARM ; /*DURATION*/
WHEN IR	ACTIVATE ALARM ; /*EVENT*/
ALL 5 MIN UNTIL 7:3:50	ACTIVATE ALARM ; /*REP-TIME*/
ALL 5 MIN DURING 3 HRS	ACTIVATE ALARM ; /*REP-DUR*/

Functional Explanation:

- The task priority, as specified in the PRIORITY attribute, is a positive number, which provides a decision making criteria for the operating system when mutually exclusive tasks are activated (see para. 15.3).
 - When the ACTIVATE statement is executed the task with the priority stated in the PRIORITY attribute will be executed.
 - When the PRIORITY attribute is missing, the task with the priority specified in its task declaration will be started.
 - The ACTIVATE statement without a SCHEDULE-1 condition immediately starts the dormant task specified in IDENTIFIER.
- Note: The ACTIVATE statement only affects dormant tasks.
- The ACTIVATE statement with a SCHEDULE-1 condition alters the state of the specified task to "runnable" only when the SCHEDULE-1 condition is reached (see para. 15). The SCHEDULE-1 condition of an activated task can be invalidated using the PREVENT statement.

16.2 Suspension of Tasks

Using the SUSPEND statement, a task can inhibit either itself or another task. The SUSPEND statement causes the following state transitions to take place:

"running"	→	"suspended"
"runnable"	→	"suspended"
"suspended"	→	"suspended"

Syntax:

SUSPEND-STATEMENT:
'SUSPEND' [IDENTIFIER].

Examples:

```
SUSPEND ALARM;
SUSPEND ; /*self-suspension*/
```

Functional Explanation:

- The IDENTIFIER specifies the task name.
- If no task name is specified then the task which executes the SUSPEND statement, will change its own state to “suspended”.
- The execution of the SUSPEND statement has no effect if the specified (other) task is already in a “suspended” state.

16.3 Continuation of Tasks

A suspended task can be continued, either conditionally or unconditionally, using the CONTINUE statement. The statement causes a change in state from “suspended” to “runnable”.

The format and the set of control conditions of the CONTINUE statement are similar to those required for the ACTIVATE statement (see para. 16.1).

Syntax:

CONTINUE-STATEMENT: [SCHEDULE-2] 'CONTINUE' IDENTIFIER [PRIORITY].
SCHEDULE-2: 'AT' CLOCK-EXPRESSION-SEVEN / 'AFTER' DURATION-EXPRESSION-SEVEN / 'WHEN' IDENTIFIER [INTEGER-IN-BRACKETS].

Examples:

```
AT 7:3:50    CONTINUE ALARM;
AFTER 8 SEC  CONTINUE ALARM;
WHEN IR      CONTINUE ALARM;
```

Functional Explanation:

- Using the unconditional CONTINUE statement the task, specified in the identifier, will be continued with the given PRIORITY.
- If the PRIORITY is missing, then the task will be continued and its original priority will be used.
- The execution of the conditional CONTINUE statement effects the continuation of the task when the SCHEDULE-2 condition holds.
- If the SCHEDULE-2 condition is satisfied then the conditional CONTINUE statement is executed and the task continued.

16.4 Resumption of Tasks

Using the RESUME statement, a task can be suspended, but automatically continued at a certain time (time of day), or after a certain interval of time, or on the occurrence of an interrupt. The RESUME statement causes the change from “running” to “suspended” state to be performed. The set of resumption conditions is in accordance with the time-dependent scheduling mechanism (see para. 15.1).

Syntax:

RESUME-STATEMENT: [SCHEDULE-2] 'RESUME' [PRIORITY].
SCHEDULE-2: 'AT' CLOCK-EXPRESSION-SEVEN / 'AFTER' DURATION-EXPRESSION-SEVEN / 'WHEN' IDENTIFIER [INTEGER-IN-BRACKETS].

Examples:

```

AT 7:3:50    RESUME;
AFTER 8 SEC  RESUME PRIO 5;
WHEN IR      RESUME;

```

Functional Explanation:

- The RESUME statement causes an immediate suspension and then a conditional continuation of the task with the SCHEDULE-2 and PRIORITY the same as in the RESUME statement.

16.5 Termination of Tasks

A task is terminated either on completion of its execution or by using a TERMINATE statement. In both cases the task is transformed to the "dormant" state, irrespective of its actual state.

Syntax:

TERMINATE STATEMENT: 'TERMINATE' [IDENTIFIER].

Examples:

```

TERMINATE;
TERMINATE ALARM;

```

Functional Explanation:

- IDENTIFIER specifies the task name.
- If no task name is specified, the task in which the TERMINATE statement is executed, will itself be terminated.
- The TERMINATE statement will have no effect on a dormant task.

16.6 Prevention of SCHEDULE Conditions

As stated previously, various execution conditions (SCHEDULE-1 or SCHEDULE-2, see para. 15) can be assigned to a given task, whenever tasks in a "running" state are executing ACTIVATE, CONTINUE or RESUME statements on a given task. There is no intermediate buffering of these conditions. Therefore the last condition specified will be prevalent.

The conditions associated with a particular task, arising from ACTIVATE, CONTINUE or RESUME statements, can be cancelled by using the PREVENT statement.

Syntax:

PREVENT-STATEMENT: 'PREVENT' [IDENTIFIER].

Example: PREVENT ALARM;

Functional Explanation:

- IDENTIFIER specifies the task name.
- If no task name is specified, the task executing the PREVENT statement cancels its own conditions.

17. SYNCHRONIZATION OF TASKS**17.1 Semaphore Variables**

Normally tasks execute their instruction independently (asynchronously) from each other. Sometimes, however, the execution of instructions must be synchronized between the various tasks.

To synchronize tasks this PEARL subset provides a special data type – semaphore variables – and specific operations to manipulate them.

Semaphores are ideal for the timing synchronization requirements as they exist whenever data has to be transferred between tasks. Using the REQUEST operation the transmission of data from a producing task can be delayed. Using the RELEASE operation a delayed task can be continued again.

REQUEST and RELEASE operations for lists of semaphores are not permitted.

17.2 Declaration and Specification of Semaphores

Semaphores can only be used with the REQUEST and RELEASE operations after they have been properly declared. Semaphores from external modules can only be used if they have been declared as GLOBAL in their declaration, and specified fully in the external module.

Syntax:

SEMA-DECLARATION: ('DECLARE' / 'DCL') (ONE-IDENTIFIER-OR-LIST 'SEMA' [GLOBAL-RESIDENT-ATTRIBUTE]// ';') ';'
SEMA-SPECIFICATION: ('SPECIFY' / 'SPC') (ONE-IDENTIFIER-OR-LIST 'SEMA' [GLOBAL-ATTRIBUTE]// ';') ';'

Examples:

```

DECLARE SYN SEMA; /*Declaration of a semaphore called SYN*/
DCL A SEMA GLOBAL; /*Declaration of semaphore A which can be used in external modules*/
SPC A SEMA GLOBAL (MOD3); /*Specification of semaphore A which is declared in module
MOD3*/

```

Functional Explanation:

- ONE-IDENTIFIER-OR-LIST specifies a list of semaphores.
- The GLOBAL-RESIDENT-ATTRIBUTE specifies that the sema may be used from external modules.
- The GLOBAL-ATTRIBUTE specifies the module in which the sema has been fully declared.

17.3 Semaphore Operations

The value of a semaphore must be an integer. At their declaration semaphores have an initial value of zero. The value of a semaphore can only be changed using a semaphore operation.

Syntax:

SYNCHRONIZATION: ('REQUEST' / 'RELEASE') IDENTIFIER.

Examples:

```

REQUEST SYN;
RELEASE A;

```

Functional Explanation:

- IDENTIFIER specifies a semaphore.
- The value of the semaphore variable is decremented by 1 by the REQUEST operation if the result is not negative. The task is not interrupted. If the result would have been negative, the operation is delayed, and the calling task suspended, until the value of the sema variable becomes positive.
- The RELEASE operation increments the value of the sema variable by 1.

APPENDIX 1

DESCRIPTION OF THE META-LANGUAGE USED TO REPRESENT THE SYNTAX
OF THE PEARL-SUBSET FOR AVIONIC APPLICATIONS

The meta-language for the definition of the syntax of programming languages as described in the following appendix, shows only minor differences from BNF. For the most part it follows the notation introduced by Deremer and Frank.

A.1 SYMBOLS OF THE META-LANGUAGE

Each symbol of the meta-language vocabulary is represented by either a designator or a literal.

A.1.1 Designators

A designator is a sequence of letters, numbers and hyphens beginning and ending with a letter or a number. Two consecutive hyphens are not permitted.

A.1.2 Literals

Literals are self-defining character strings. The characters making up a literal form a terminal symbol, and in any input text, are expected in the form defined by the literal.

A literal is enclosed in single quotes. If a single quote is included in the literal it is preceded by another quote mark.

A.1.3 Operators

The following special characters are used to represent operators:

:
*
/
//
+
\$
(
)
[
]
.

A.1.4 Use of Space Character(s)

Within a meta-program, spaces are used to separate consecutive symbols. These may be left out when consecutive symbols can be identified without the insertion of spaces.

A.2 PRODUCTION RULES

A meta-program begins with a key-word GRAMMATIK-REGELN and ends with a key-word FINIS. Between the two key-words a series of production rules are defined which are then used to describe the syntax of the programming language to be represented.

A.2.1 Construction of a Production Rule

A production rule consists of

- A designator (name of the rule) used to refer to the rule body in subsequent production rules,
- a colon, to separate the name of the rule from the rule body,
- a rule body which specifies its syntax, and
- a full-stop to terminate the production rule.

A meta-program is defined as complete when:

- Each designator of a rule body identifies one and only one body of the rule,
- one and only one rule name exists which is used in no rule body (start symbol).

A.2.2 Alternatives

Alternatives, within a rule body, are separated by means of oblique strokes (/).

Example:

$$A : B / C .$$

A is defined as either B or C.

A.2.3 Options

If one alternative symbol string is merely a longer representation of the other alternative (i.e. The first parts of each alternative are identical.) the remainder of the longer string can be enclosed in square brackets to indicate that it is optional.

Example: Using square brackets the production rule

$$A : B C D / B .$$

may be represented as

$$A : B [C D] .$$

A.2.4 Strings

When a rule body contains a string of symbols which must occur at least once then this string of symbols can be followed by an asterisk indicating that the symbol string may be repeated.

Example: The production rule

$$A : B [A] .$$

may also be represented as

$$A : B * .$$

If the symbol string can be "empty", then instead of the asterisk a dollar sign is used.

Example:

$$A : [B [A]] .$$

is equivalent to:

$$A : B \$.$$

A.2.5 Subdivision

The operator, //, can be used to indicate that the symbol chain enclosed in quotes appears in a rule body between successive occurrence of the symbol chain before the operator (The "//" operator signifies "subdivided by"). This explanation is clarified by the following example:

Example: The rule

$$A : B / B ' ' A .$$

produces:

B
B,B
B,B,B
.
.
.

The same rule can also be represented as:

$$A : B // ' ' .$$

A.2.6 Optional Chaining

The + operator is used to indicate that either the symbols in the rule body may occur alone, or they can both occur together. However, when they both occur together the sequence in which they occur in the rule body must be observed.

Example: $A : B [C] / C$

is equivalent with:

$A : B + C .$

A.2.7 Use of Brackets

Whenever a rule name occurs within a rule body then it may be replaced by its related rule body. Whenever this rule body consists of more than a single symbol it must be enclosed in round brackets when it is substituted. Whenever a rule name is replaced in all rules in which it occurs then the related production rule may be deleted.

Example: If

$X : Y Z .$

then the following definitions of A are equivalent:

$A : B X* .$

$A : B (Y Z)* .$

Since both the operators, // and +, are left associative there are many rules in which the parentheses enclosing the operators can be omitted.

Example: The following three definitions of A are equivalent:

$A : X // D . \quad X : B // C .$

$A : (B // C) // D .$

$A : B // C // D .$

APPENDIX 2

SYNTAX OF THE PEARL-SUBSET FOR AVIONIC APPLICATIONS

(prepared by GPP)

The following chapter deals with the production rules of the PEARL-Subset for AVIONIC applications. It contains cross-reference-lists which provide a quick and easy reference to the different rule-names and literals which occur within the production rules.

These cross-reference-lists make it easy to find the occurrence of a production-rule or a literal. Furthermore, they provide a very convenient overview of the chosen subset, and they demonstrate the completeness of the representation.

A cross-reference-list is in two parts, a list of the rule-names, and a list of the literals that occur in the production rules of the syntax.

In the rule-names section, there are three fields, occurring under the headings: SYMBOL, DEFINITION, and VERWENDUNG respectively.

SYMBOL: gives the name of the rule.

DEFINITION: gives the line-number (which appears to the text of each rule in the production listing) of the production that defines the rule.

VERWENDUNG: gives the line-number of each use of the rule in other rules of the Subset.

In the literals section, the symbol field and the Verwendung field give the literal and its uses respectively. No definitions are given for literals.

PEARL-AVIONIC-Subset

1. List of Syntax Rules

```

1  GRAMMATIK-REGELN
2
3
4
5  CONSTANT-DENOTATION:
6      INTEGER-CONSTANT-DENOTATION /
7      REAL-CONSTANT-DENOTATION /
8      CHARACTER-STRING-CONSTANT-DENOTATION /
9      BIT-STRING-CONSTANT-DENOTATION /
10     CLOCK-CONSTANT-DENOTATION /
11     DURATION-CONSTANT-DENOTATION.
12
13
14
15  INTEGER-CONSTANT-DENOTATION:  DIGIT*.
16
17
18
19  DIGIT:
20     OCTAL-DIGIT / '8' / '9'.
21
22
23
24  OCTAL-DIGIT:
25     '0' / '1' / '2' / '3' /
26     '4' / '5' / '6' / '7'.
27
28
29
30  REAL-CONSTANT-DENOTATION:
31     ( DIGIT* '.' DIGIT* / DIGIT* '.' )
32     [EXPONENTIAL-PART] /
33     DIGIT* EXPONENTIAL-PART.
34
35
36
37  EXPONENTIAL-PART:
38     'E' ['+' / '-'] DIGIT*.
39
40
41
42  CHARACTER-STRING-CONSTANT-DENOTATION:
43     '...' [STRING-CHARACTER *] '...'.
44
45
46
47  STRING-CHARACTER:
48     LETTER / DIGIT /
49     ' ' / '+' / '-' / '*' / '/' /
50     '<' / '>' / '#' / '$' / ':' /
51     '.' / ',' / ';' / '=' / '!' / '?' /

```


PEARL-AVIONIC-Subse

```

52      '=' / '<' / '>'.
53
54
55
56  BIT-STRING-CONSTANT-DENOTATION:
57      '""' BINARY-DIGIT* '""' ( 'B' / 'B1' ) /
58      '""' OCTAL-DIGIT* '""' 'B3' /
59      '""' SEDECIMAL-DIGIT* '""' 'B4'.
60
61
62
63  BINARY-DIGIT: 'B' / '1'.
64
65
66
67  LETTER:
68      'A' / 'B' / 'C' / 'D' / 'E' / 'F' /
69      'G' / 'H' / 'I' / 'J' / 'K' / 'L' /
70      'M' / 'N' / 'O' / 'P' / 'Q' / 'R' /
71      'S' / 'T' / 'U' / 'V' / 'W' / 'X' /
72      'Y' / 'Z'.
73
74
75
76
77  SEDECIMAL-DIGIT:
78      DIGIT / 'A' / 'B' / 'C' /
79      'D' / 'E' / 'F'.
80
81
82
83  CLOCK-CONSTANT-DENOTATION:
84      DIGIT* ':'
85      DIGIT* ':'
86      DIGIT*.
87
88
89
90
91  DURATION-CONSTANT-DENOTATION:
92      DIGIT* 'HRS'
93      +
94      DIGIT* 'MIN'
95      +
96      DIGIT* 'SEC'
97      +
98      DIGIT* 'MSEC'.
99
100
101
102  IDENTIFIER:
103      LETTER ( LETTER / DIGIT )$.
104
105
106
107  DEVICE-IDENTIFIER: IDENTIFIER.

```

PEARL-AVIONIC-Subset

```
108
109
110
111     FILE-IDENTIFIER: IDENTIFIER.
112
113
114
115     FORMAT-PROCEDURE-IDENTIFIER: IDENTIFIER.
116
117
118
119     GLOBAL-QUALIFIER: IDENTIFIER.
120
121
122
123     LABEL-IDENTIFIER: IDENTIFIER.
124
125
126
127     PARAMETER-IDENTIFIER: IDENTIFIER.
128
129
130
131     SELECTOR-IDENTIFIER: IDENTIFIER.
132
133
134
135     SIGNAL-IDENTIFIER: IDENTIFIER.
136
137
138
139     ONE-IDENTIFIER-OR-LIST:
140         IDENTIFIER / IDENTIFIER-LIST.
141
142
143
144     IDENTIFIER-LIST:
145         '(' ( IDENTIFIER // ',' ) ')'.
146
147
148
149     ONE-SYMBOL-OR-LIST:
150         SYMBOL /
151         '(' ( SYMBOL // ',' ) ')'.
152
153
154
155     SYMBOL:
156         IDENTIFIER
157         ( '.' IDENTIFIER / EXPRESSION-SEVEN-PACK )$.
158
159
160
161     INTEGER-CONSTANT-LIST:
162         '(' ( INTEGER-CONSTANT-DENOTATION // ',' ) ')'.
163
```

PEARL-AVIONIC-Subset

```

164
165
166 INTEGER-IN-BRACKETS:
167     '(' INTEGER-CONSTANT-DENOTATION ')'
168
169
170
171 PROGRAM-MODULE:
172     'MODULE' ['(' GLOBAL-QUALIFIER ')'] ';'
173     ( 'SYSTEM' ';' ( CONNECTION ';' ) *
174       +
175       'PROBLEM' ';' DECLARATIONS-AND-SPECIFICATIONS )
176     'MODEND' ';'
177
178
179
180 DECLARATIONS-AND-SPECIFICATIONS:
181     ( LENGTH-DECLARATION ';' ) $
182     +
183     ( GLOBAL-SPECIFICATION ';' ) $
184     +
185     ( MODULE-IDENTIFIER-DECLARATION ';' ) $
186     +
187     ( LABEL-IDENTIFIER ';' ( PROCEDURE-DECLARATION /
188       TASK-DECLARATION ) ';' ) $.
189
190
191
192 LENGTH-DECLARATION:
193     'LENGTH'
194     ( ( 'FIXED' / 'FLOAT' )
195       '(' INTEGER-CONSTANT-DENOTATION ')' // ',' ).
196
197
198
199 GLOBAL-SPECIFICATION:
200     ( 'SPECIFY' / 'SPC' )
201     ( ONE-IDENTIFIER-OR-LIST
202       GLOBAL-MODE
203       [GLOBAL-ATTRIBUTE] // ',' ).
204
205
206
207 GLOBAL-MODE:
208     MODULE-MODE /
209     PROCEDURE-MODE /
210     TASK-MODE.
211
212
213
214 MODULE-MODE:
215     LOCAL-MODE /
216     'SEMA' /
217     DEVICE-OR-INTERRUPT-OR-SIGNAL-MODE /
218     FILE-MODE.
219

```

PEARL-AVIONIC-Subset

```

220
221
222 LOCAL-MODE:
223     [INTEGER-CONSTANT-LIST]
224     ['INV'] ( SIMPLE-MODE /
225              STRUCTURE-MODE ).
226
227
228
229 SIMPLE-MODE:
230     'CLOCK' / 'DUR' /
231     ( 'BIT' / 'CHAR' /
232       'FIXED' / 'FLOAT' ) [INTEGER-IN-BRACKETS] .
233
234
235
236 STRUCTURE-MODE:
237     'STRUCT' '('
238       ( ONE-IDENTIFIER-OR-LIST SIMPLE-MODE // ',' )
239       ')'.
240
241
242
243 DEVICE-OR-INTERRUPT-OR-SIGNAL-MODE:
244     [INTEGER-IN-BRACKETS]
245     ( 'INTERRUPT' / 'IRUPT' /
246       'SIGNAL' /
247       DEVICE-MODE ).
248
249
250
251 DEVICE-MODE:
252     ( 'DEVICE' / 'DVC' ) DEVICE-TYPE.
253
254
255
256 DEVICE-TYPE:
257     ['INTERNAL']
258     ( 'SOURCE' / 'SINK' / 'SOUSI' )
259     ['DIRECT']
260     [SIMPLE-MODE] .
261
262
263
264 FILE-MODE:
265     'FILE' [FILE-TYPE] .
266
267
268
269 FILE-TYPE:
270     FILE-CLASS
271     FILE-USAGE
272     FILE-ACCESS
273     [FILE-CONTROL]
274     [FILE-CONTINUATION]
275     [FILE-DIMENSION]

```


PEARL-AVIONIC-Subset

```

276      FILE-CONTENTS.
277
278
279
280      FILE-CLASS:
281          'ANALOG' / 'EXTERNAL' / 'INTERNAL'.
282
283
284
285      FILE-USAGE:
286          'SOURCE' / 'SINK' / 'SOUSI'.
287
288
289
290      FILE-ACCESS:
291          'DIRECT' / 'FORWARD'.
292
293
294
295      FILE-CONTROL:
296          'SIGNAL' '(' (SIGNAL-IDENTIFIER // ',' ) ')'.
297
298
299
300      FILE-CONTINUATION:
301          ( ['STREAM'] / 'NOSTREAM' )
302          ['NOCYCL'].
303
304
305
306      FILE-DIMENSION:
307          '(' ( [INTEGER-CONSTANT-DENOTATION] // ',' ) ')'.
308
309
310
311      FILE-CONTENTS:
312          LOCAL-MODE.
313
314
315
316      PROCEDURE-MODE:
317          'ENTRY' [ '(' ( PARAMETER-MODE // ',' ) ') ' ]
318                  [RESULT-ATTRIBUTE].
319
320
321
322      PARAMETER-MODE:
323          [ '(' ( INTEGER-CONSTANT-DENOTATION // ',' ) ') ' ]
324          ( ( ['INV'] ( SIMPLE-MODE /
325                  STRUCTURE-MODE ) ['IDENTICAL' / 'IDENT'] ),
326
327          DEVICE-MODE ) /
328      FILE-MODE.
329
330

```

PEARL-AVIONIC-Subset

```

331 RESULT-ATTRIBUTE:
332   'RETURNS' '(' SIMPLE-MODE ')'.
333
334
335
336 GLOBAL-ATTRIBUTE:
337   'GLOBAL' ['(' GLOBAL-QUALIFIER ')'].
338
339
340
341 TASK-MODE:
342   'TASK'.
343
344
345
346 MODULE-IDENTIFIER-DECLARATION:
347   ( 'DECLARE' / 'DCL' )
348     ( ONE-IDENTIFIER-OR-LIST
349       MODULE-MODE
350         [ ( 'INITIAL' / 'INIT' ) '<'
351           ( EXPRESSION-SEVEN // ',' ) '>' /
352           ( 'IDENTICAL' / 'IDENT' ) SYMBOL ]
353         [GLOBAL-RESIDENT-ATTRIBUTE]
354         // ',' ) .
355
356
357
358 GLOBAL-RESIDENT-ATTRIBUTE:
359   GLOBAL-ATTRIBUTE ['RESIDENT'] .
360
361
362
363 PROCEDURE-DECLARATION:
364   'PROC' ['(' ( ONE-IDENTIFIER-OR-LIST
365                 PARAMETER-MODE // ',' ) ')']
366     [RESULT-ATTRIBUTE]
367     [GLOBAL-RESIDENT-ATTRIBUTE]
368     ['REENTRANT'] ','
369     BLOCK-TAIL.
370
371
372
373 BLOCK-TAIL:
374   ( LOCAL-IDENTIFIER-DECLARATION ';' ) $
375   STATEMENT $
376   'END'.
377
378
379
380 LOCAL-IDENTIFIER-DECLARATION:
381   ( 'DECLARE' / 'DCL' )
382     ( ONE-IDENTIFIER-OR-LIST
383       LOCAL-MODE
384         [ ( 'INITIAL' / 'INIT' ) '<'
385           ( EXPRESSION-SEVEN // ',' ) '>' /
386           ( 'IDENTICAL' / 'IDENT' ) SYMBOL ]

```

PEARL-AVIONIC-Subset

```
387          // ' , ' ) .
388
389
390
391 TASK-DECLARATION:
392     'TASK' [PRIORITY]
393             [GLOBAL-RESIDENT-ATTRIBUTE]
394             BLOCK-TAIL.
395
396
397
398 PRIORITY:
399     ( 'PRIORITY' / 'PRIO' )
400     INTEGER-CONSTANT-DENOTATION.
401
402
403
404 STATEMENT:
405     ( ( LABEL-IDENTIFIER ':' ) $ /
406       ( 'ON' SIGNAL-IDENTIFIER
407         [INTEGER-IN-BRACKETS] ':' ) $ )
408     STATEMENT-BODY ' ; ' .
409
410
411
412 STATEMENT-BODY:
413     ASSIGNMENT /
414     BEGIN-BLOCK /
415     SEQUENTIAL-CONTROL /
416     PARALLEL-CONTROL /
417     SYNCHRONIZATION /
418     INTERRUPTION /
419     TRANSPUT.
420
421
422
423 ASSIGNMENT:
424     SYMBOL ( ':' / '=' ) EXPRESSION-SEVEN.
425
426
427
428 BIT-ONE-EXPRESSION-SEVEN: EXPRESSION-SEVEN.
429
430
431
432 CLOCK-EXPRESSION-SEVEN: EXPRESSION-SEVEN.
433
434
435
436 DURATION-EXPRESSION-SEVEN: EXPRESSION-SEVEN.
437
438
439
440 INTEGER-EXPRESSION-SEVEN: EXPRESSION-SEVEN.
441
442
```

PEARL-AVIONIC-Subset

```

443
444     EXPRESSION-SEVEN-PACK:
445         '(' < EXPRESSION-SEVEN // ',' > ')'.
446
447
448
449     EXPRESSION-SEVEN:
450         EXPRESSION-SIX // PRECEDENCE-SEVEN-OPERATOR.
451
452
453
454     EXPRESSION-SIX:
455         EXPRESSION-FIVE // PRECEDENCE-SIX-OPERATOR.
456
457
458
459     EXPRESSION-FIVE:
460         EXPRESSION-FOUR // PRECEDENCE-FIVE-OPERATOR.
461
462
463
464     EXPRESSION-FOUR:
465         EXPRESSION-THREE // PRECEDENCE-FOUR-OPERATOR.
466
467
468
469     EXPRESSION-THREE:
470         EXPRESSION-TWO // PRECEDENCE-THREE-OPERATOR.
471
472
473
474     EXPRESSION-TWO:
475         EXPRESSION-ONE // PRECEDENCE-TWO-OPERATOR.
476
477
478
479     EXPRESSION-ONE:
480         PRIMITIVE-EXPRESSION
481         [PRECEDENCE-ONE-OPERATOR EXPRESSION-ONE] /
482         MONADIC-OPERATOR EXPRESSION-ONE.
483
484
485
486     PRIMITIVE-EXPRESSION:
487         CONSTANT-DENOTATION /
488         SYMBOL /
489         '(' EXPRESSION-SEVEN ')'.
490
491
492
493     PRECEDENCE-SEVEN-OPERATOR:     'OR' .
494
495
496
497     PRECEDENCE-SIX-OPERATOR:     'AND' .
498

```


PEARL-AVIONIC-Subset

```
499
500
501 PRECEDENCE-FIVE-OPERATOR:  '=' / 'EQ' /
502                             '/=' / 'NE' .
503
504
505
506 PRECEDENCE-FOUR-OPERATOR:  '<' / 'LT' /
507                             '>' / 'GT' /
508                             '<=' / 'LE' /
509                             '>=' / 'GE' .
510
511
512
513 PRECEDENCE-THREE-OPERATOR:  '+' /
514                             '-' /
515                             '<>' / 'CAT' /
516                             '><' / 'CSHIFT' /
517                             'SHIFT' .
518
519
520
521 PRECEDENCE-TWO-OPERATOR:    '*' /
522                             '/' /
523                             '//' .
524
525
526
527 PRECEDENCE-ONE-OPERATOR:    '**' /
528                             'FIT' .
529
530
531
532 MONADIC-OPERATOR:          '+' /
533                             '-' /
534                             'NOT' /
535                             'FLOAT' /
536                             'FIX' /
537                             'CHAR' /
538                             'BIT' .
539
540
541
542 BEGIN-BLOCK:
543     'BEGIN'  ';'  BLOCK-TAIL.
544
545
546
547 SEQUENTIAL-CONTROL:
548     SKIP-STATEMENT /
549     GOTO-STATEMENT /
550     CONDITIONAL-STATEMENT /
551     CASE-STATEMENT /
552     REPEAT-STATEMENT /
553     CALL-STATEMENT /
554     RETURN-STATEMENT .
```

PEARL-AVIONIC-Subset

```
555
556
557
558     SKIP-STATEMENT:  'SKIP'.
559
560
561
562     GOTO-STATEMENT:
563         'GOTO' LABEL-IDENTIFIER.
564
565
566
567     CONDITIONAL-STATEMENT:
568         'IF' BIT-ONE-EXPRESSION-SEVEN
569         'THEN' STATEMENT*
570         ['ELSE' STATEMENT*] 'FIN'.
571
572
573
574     CASE-STATEMENT:
575         'CASE' INTEGER-EXPRESSION-SEVEN
576         ( 'ALT' STATEMENT $ ) *
577         ['OUT' STATEMENT*] 'FIN'.
578
579
580
581     REPEAT-STATEMENT:
582         ['FOR' IDENTIFIER
583         +
584         'FROM' EXPRESSION-SEVEN
585         +
586         'BY' EXPRESSION-SEVEN
587         +
588         'TO' EXPRESSION-SEVEN
589         +
590         'WHILE' EXPRESSION-SEVEN]
591         'REPEAT' STATEMENT * 'END'.
592
593
594
595     CALL-STATEMENT:
596         'CALL' IDENTIFIER [EXPRESSION-SEVEN-PACK].
597
598
599
600     RETURN-STATEMENT:
601         'RETURN' ['(' EXPRESSION-SEVEN ')'].
602
603
604
605     PARALLEL-CONTROL:
606         ACTIVATE-STATEMENT /
607         SUSPEND-STATEMENT /
608         CONTINUE-STATEMENT /
609         RESUME-STATEMENT /
610         TERMINATE-STATEMENT /
```

PEARL-AVIONIC-Subset

```
611      PREVENT-STATEMENT.  
612  
613  
614  
615      ACTIVATE-STATEMENT:  
616          [SCHEDULE-1] 'ACTIVATE' IDENTIFIER [PRIORITY] .  
617  
618  
619  
620      SUSPEND-STATEMENT:  
621          'SUSPEND' [IDENTIFIER] .  
622  
623  
624  
625      CONTINUE-STATEMENT:  
626          [SCHEDULE-2] 'CONTINUE' IDENTIFIER [PRIORITY] .  
627  
628  
629  
630      RESUME-STATEMENT:  
631          SCHEDULE-2 'RESUME' .  
632  
633  
634  
635      PREVENT-STATEMENT:  
636          'PREVENT' [IDENTIFIER] .  
637  
638  
639  
640      TERMINATE-STATEMENT:  
641          'TERMINATE' [IDENTIFIER] .  
642  
643  
644  
645      SCHEDULE-1:  
646          SCHEDULE-2  
647          +  
648          REPETITION-OPTION.  
649  
650  
651  
652      SCHEDULE-2:  
653          'AT' CLOCK-EXPRESSION-SEVEN /  
654          'AFTER' DURATION-EXPRESSION-SEVEN /  
655          'WHEN' IDENTIFIER [INTEGER-IN-BRACKETS] .  
656  
657  
658  
659      REPETITION-OPTION:  
660          'ALL' DURATION-EXPRESSION-SEVEN  
661          ['UNTIL' CLOCK-EXPRESSION-SEVEN /  
662          'DURING' DURATION-EXPRESSION-SEVEN] .  
663  
664  
665  
666      SYNCHRONIZATION:
```

PEARL-AVIONIC-Subset

```
667      ( 'REQUEST' / 'RELEASE' ) IDENTIFIER.
668
669
670
671  INTERRUPTION:
672      ( 'ENABLE' /
673        'DISABLE' /
674        'TRIGGER' /
675        'INDUCE' ) SYMBOL.
676
677
678
679  TRANSPUT:
680      CREATE-STATEMENT /
681      DELETE-STATEMENT /
682      OPEN-STATEMENT /
683      CLOSE-STATEMENT /
684      GET-STATEMENT /
685      PUT-STATEMENT /
686      SEE-STATEMENT /
687      DRAW-STATEMENT /
688      TAKE-STATEMENT /
689      SEND-STATEMENT.
690
691
692
693  CREATE-STATEMENT:
694      'CREATE'
695      ( TITLE / FILE-IDENTIFIER )
696      UPON.
697
698
699
700  TITLE:
701      'TITLE' '(' CHARACTER-STRING-CONSTANT-DENOTATION ')'
702      [FILE-TYPE].
703
704
705
706  UPON:
707      'UPON' DEVICE-IDENTIFIER.
708
709
710
711  DELETE-STATEMENT:
712      'DELETE'
713      ( TITLE / FILE-IDENTIFIER ).
714
715
716
717  OPEN-STATEMENT:
718      'OPEN' FILE-IDENTIFIER
719      [[TITLE] UPON].
720
721
722
```


PEARL-AVIONIC-Subset

```
723 CLOSE-STATEMENT:
724     'CLOSE' FILE-IDENTIFIER.
725
726
727
728 GET-STATEMENT:
729     'GET' 'FROM' FILE-ELEMENT
730         'TO' ONE-SYMBOL-OR-LIST
731         ['THRU' STANDARD-FORMAT-LIST].
732
733
734
735 FILE-ELEMENT:
736     FILE-IDENTIFIER ['AT' '(' (FILE-INDEX // ',' ) ')'].
737
738
739
740 FILE-INDEX:
741     ['HERE' ('+' / '-') ] INTEGER-EXPRESSION-SEVEN.
742
743
744
745 STANDARD-FORMAT-LIST:
746     [INTEGER-IN-BRACKETS] ( C-FORMAT-ELEMENT /
747                             C-FORMAT-LIST ) .
748
749
750
751 C-FORMAT-LIST:
752     '(' ( STANDARD-FORMAT-LIST // ',' ) ')'.
753
754
755
756 C-FORMAT-ELEMENT:
757     C-FORMAT-CONTROL-ELEMENT /
758     C-FORMAT-DATA-ELEMENT.
759
760
761
762 C-FORMAT-CONTROL-ELEMENT:
763     ( 'LINE' /
764       'X' /
765       'SKIP' /
766       'PAGE' /
767       'COL' ) [INTEGER-IN-BRACKETS].
768
769
770
771 C-FORMAT-DATA-ELEMENT:
772     ( 'A' / 'B' / 'B3' / 'B4' ) [INTEGER-IN-BRACKETS] /
773     ( 'E' / 'F' ) '(' INTEGER-CONSTANT-DENOTATION
774         [',' OPTIONAL] ')' /
775     ( 'D' / 'T' ) '(' [OPTIONAL] ')'.
776
777
778
```

PEARL-AVIONIC-Subset

```

779  OPTIONAL:
780      INTEGER-CONSTANT-DENOTATION
781      [' ' INTEGER-CONSTANT-DENOTATION] .
782
783
784
785  PUT-STATEMENT:
786      'PUT' 'FROM' ONE-SYMBOL-OR-LIST
787      'TO' FILE-ELEMENT
788      ['THRU' STANDARD-FORMAT-LIST] .
789
790
791
792  TAKE-STATEMENT:
793      'TAKE' 'FROM' DEVICE-ELEMENT
794      ['TO' SYMBOL]
795      ['THRU' P-OPTION] .
796
797
798
799  DEVICE-ELEMENT:
800      DEVICE-IDENTIFIER ['(' EXPRESSION-SEVEN ')'] .
801
802
803
804  P-OPTION:
805      FORMAT-PROCEDURE-IDENTIFIER
806      ['(' EXPRESSION-SEVEN-PACK ')'] .
807
808
809
810  SEND-STATEMENT:
811      'SEND' ['FROM' SYMBOL]
812      'TO' DEVICE-ELEMENT
813      ['THRU' P-OPTION] .
814
815
816
817  SEE-STATEMENT:
818      'SEE' 'FROM' FILE-ELEMENT
819      'TO' ONE-SYMBOL-OR-LIST
820      'THRU' GRAPHIC-FORMAT-LIST .
821
822
823
824  GRAPHIC-FORMAT-LIST:
825      [INTEGER-IN-BRACKETS] ( G-FORMAT-ELEMENT /
826      G-FORMAT-LIST ) .
827
828
829
830  G-FORMAT-LIST:
831      '<' < GRAPHIC-FORMAT-LIST // ' ' > ' ' .
832
833
834

```

PEARL-AVIONIC-Subset

```

835 G-FORMAT-ELEMENT:
836     'FRAME' / 'BR' / 'MAP' /
837     'XA' / 'XR' / 'XS' / 'XI' / 'XP' /
838     'YA' / 'YR' / 'YS' / 'YI'
839
840
841
842 DRAW-STATEMENT:
843     'DRAW' 'FROM' ONE-SYMBOL-OR-LIST
844     'TO' FILE-ELEMENT
845     'THRU' GRAPHIC-FORMAT-LIST.
846
847
848
849 CONNECTION:
850     SINGLE-CONNECTION TRANSFER-DIRECTION [SINGLE-CONNECTION]/
851
852     TRANSFER-DIRECTION SINGLE-CONNECTION /
853     ARRAY-CONNECTION TRANSFER-DIRECTION [GROUP-CONNECTION] /
854     [GROUP-CONNECTION] TRANSFER-DIRECTION ARRAY-CONNECTION.
855
856
857 TRANSFER-DIRECTION:
858     '<-' /
859     '<->' /
860     '->'
861
862
863
864 SINGLE-CONNECTION:
865     CONNECTION-IDENTIFICATION $
866     +
867     CONNECTION-DETERMINATION
868     +
869     CONNECTOR-DESCRIPTION.
870
871
872
873 CONNECTION-IDENTIFICATION:
874     IDENTIFIER ':'.
875
876
877
878 CONNECTION-DETERMINATION:
879     CONNECTION-TYPE
880     ['<' INTEGER-CONSTANT-DENOTATION '>'].
881
882
883
884 CONNECTION-TYPE: IDENTIFIER.
885
886
887
888 CONNECTOR-DESCRIPTION:
889     '*' ( INTEGER-CONSTANT-DENOTATION / IDENTIFIER )

```

PEARL-AVIONIC-Subset

```
890      +
891      SUBDIVIDED-CONNECTOR-DESCRIPTOR.
892
893
894
895      SUBDIVIDED-CONNECTOR-DESCRIPTOR:
896      '*' INTEGER-CONSTANT-DENOTATION
897      ',' INTEGER-CONSTANT-DENOTATION
898
899
900
901      ARRAY-CONNECTION:
902      IDENTIFIER ARRAY-BOUNDS ':'
903      +
904      CONNECTION-TYPE ARRAY-BOUNDS.
905
906
907
908      ARRAY-BOUNDS:
909      '(' INTEGER-CONSTANT-DENOTATION
910      ':' INTEGER-CONSTANT-DENOTATION ')'.
911
912
913
914      GROUP-CONNECTION:
915      CONNECTION-DETERMINATION
916      ['*' ( INTEGER-CONSTANT-DENOTATION / ARRAY-BOUNDS )
917      [SUBDIVIDED-CONNECTOR-DESCRIPTOR]] // '+'.
918
919
920
921      FINIS
```


PEARL-AVIONIC-Subset

2. Cross-Referenz-List of the Rulenames

SYMBOL	REGELNAMEN	DEFINITION	VERWENDUNG
ACTIVATE-STATEMENT	615	606	
ARRAY-BOUNDS	908	902, 904, 916	
ARRAY-CONNECTION	901	852, 853	
ASSIGNMENT	423	413	
BEGIN-BLOCK	542	414	
BINARY-DIGIT	63	57	
BIT-ONE-EXPRESSION-SEVEN	428	568	
BIT-STRING-CONSTANT-DENOTATION	56	9	
BLOCK-TAIL	373	369, 394, 543	
C-FORMAT-CONTROL-ELEMENT	762	757	
C-FORMAT-DATA-ELEMENT	771	758	
C-FORMAT-ELEMENT	756	746	
C-FORMAT-LIST	751	747	
CALL-STATEMENT	595	553	
CASE-STATEMENT	574	551	
CHARACTER-STRING-CONSTANT-DENOTATION	42	8, 701	
CLOCK-CONSTANT-DENOTATION	83	10	
CLOCK-EXPRESSION-SEVEN	432	653, 661	
CLOSE-STATEMENT	723	683	
CONDITIONAL-STATEMENT	567	550	
CONNECTION	849	173	
CONNECTION-DETERMINATION	878	867, 915	
CONNECTION-IDENTIFICATION	873	865	
CONNECTION-TYPE	884	879, 904	
CONNECTOR-DESCRIPTION	888	869	
CONSTANT-DENOTATION	5	487	
CONTINUE-STATEMENT	625	608	
CREATE-STATEMENT	693	680	
DECLARATIONS-AND-SPECIFICATIONS	180	175	
DELETE-STATEMENT	711	681	
DEVICE-ELEMENT	799	793, 812	
DEVICE-IDENTIFIER	107	707, 800	
DEVICE-MODE	251	247, 326	
DEVICE-OR-INTERRUPT-OR-SIGNAL-MODE	243	217	
DEVICE-TYPE	256	252	
DIGIT	19	15, 31, 31, 31, 33, 39, 48, 78, 84, 85, 86, 92, 94, 96, 98, 103	
DRAW-STATEMENT	842	687	
DURATION-CONSTANT-DENOTATION	91	11	
DURATION-EXPRESSION-SEVEN	436	654, 660, 662	
EXPONENTIAL-PART	37	32, 33	
EXPRESSION-FIVE	459	455	
EXPRESSION-FOUR	464	460	
EXPRESSION-ONE	479	475, 481, 482	
EXPRESSION-SEVEN	449	351, 385, 424, 428, 432, 436, 440, 445, 489, 584, 586, 588, 590, 601, 800	
EXPRESSION-SEVEN-PACK	444	157, 596, 806	
EXPRESSION-SIX	454	450	

PEARL-AVIONIC-Subset

EXPRESSION-THREE	469	465
EXPRESSION-TWO	474	470
FILE-ACCESS	290	272
FILE-CLASS	290	270
FILE-CONTENTS	311	276
FILE-CONTINUATION	300	274
FILE-CONTROL	295	273
FILE-DIMENSION	306	275
FILE-ELEMENT	735	729, 787, 818, 844
FILE-IDENTIFIER	111	695, 713, 718, 724, 736
FILE-INDEX	740	736
FILE-MODE	264	218, 327
FILE-TYPE	269	265, 702
FILE-USAGE	285	271
FORMAT-PROCEDURE-IDENTIFIER	115	805
G-FORMAT-ELEMENT	835	825
G-FORMAT-LIST	830	826
GET-STATEMENT	728	684
GLOBAL-ATTRIBUTE	336	203, 359
GLOBAL-MODE	207	202
GLOBAL-QUALIFIER	119	172, 337
GLOBAL-RESIDENT-ATTRIBUTE	358	353, 367, 393
GLOBAL-SPECIFICATION	199	183
GOTO-STATEMENT	562	549
GRAPHIC-FORMAT-LIST	824	820, 831, 845
GROUP-CONNECTION	914	852, 853
IDENTIFIER	102	107, 111, 115, 119, 123, 127, 131, 135, 140, 145, 156, 157, 582, 596, 616, 621, 626, 636, 641, 655, 667, 874, 884, 889, 902
IDENTIFIER-LIST	144	140
INTEGER-CONSTANT-DENOTATION	15	6, 162, 167, 195, 307, 323, 400, 773, 780, 781, 880, 889, 896, 897, 909, 910, 916
INTEGER-CONSTANT-LIST	161	223
INTEGER-EXPRESSION-SEVEN	440	575, 741
INTEGER-IN-BRACKETS	166	232, 244, 407, 655, 746, 767, 772, 825
INTERRUPTION	671	410
LABEL-IDENTIFIER	123	187, 405, 563
LENGTH-DECLARATION	192	181
LETTER	67	48, 103, 103
LOCAL-IDENTIFIER-DECLARATION	380	374
LOCAL-MODE	222	215, 312, 383
MODULE-IDENTIFIER-DECLARATION	346	185
MODULE-MODE	214	208, 349
MONADIC-OPERATOR	532	482
OCTAL-DIGIT	24	20, 58
ONE-IDENTIFIER-OR-LIST	139	201, 238, 348, 364, 382
ONE-SYMBOL-OR-LIST	149	730, 786, 819, 843
OPEN-STATEMENT	717	682
OPTIONAL	779	774, 775
P-OPTION	804	795, 813
PARALLEL-CONTROL	605	416
PARAMETER-IDENTIFIER	127	NICHT VERWENDET
PARAMETER-MODE	322	317, 365

PEARL-AVIONIC-Subse1

PRECEDENCE-FIVE-OPERATOR	501	460
PRECEDENCE-FOUR-OPERATOR	506	465
PRECEDENCE-ONE-OPERATOR	527	481
PRECEDENCE-SEVEN-OPERATOR	493	450
PRECEDENCE-SIX-OPERATOR	497	455
PRECEDENCE-THREE-OPERATOR	513	470
PRECEDENCE-TWO-OPERATOR	521	475
PREVENT-STATEMENT	635	611
PRIMITIVE-EXPRESSION	486	480
PRIORITY	398	392, 616, 626
PROCEDURE-DECLARATION	363	187
PROCEDURE-MODE	316	209
PROGRAM-MODULE	171	NICHT VERWENDET
PUT-STATEMENT	785	685
REAL-CONSTANT-DENOTATION	30	7
REPEAT-STATEMENT	581	552
REPETITION-OPTION	659	648
RESULT-ATTRIBUTE	331	318, 366
RESUME-STATEMENT	630	609
RETURN-STATEMENT	600	554
SCHEDULE-1	645	616
SCHEDULE-2	652	626, 631, 646
SEDECIMAL-DIGIT	77	59
SEE-STATEMENT	817	686
SELECTOR-IDENTIFIER	131	NICHT VERWENDET
SEND-STATEMENT	810	689
SEQUENTIAL-CONTROL	547	415
SIGNAL-IDENTIFIER	135	296, 406
SIMPLE-MODE	229	224, 238, 260, 324, 332
SINGLE-CONNECTION	864	950, 850, 851
SKIP-STATEMENT	558	548
STANDARD-FORMAT-LIST	745	731, 752, 788
STATEMENT	404	375, 569, 570, 576, 577, 591
STATEMENT-BODY	412	408
STRING-CHARACTER	47	43
STRUCTURE-MODE	236	225, 325
SUBDIVIDED-CONNECTOR-DESCRIPTOR	895	891, 917
SUSPEND-STATEMENT	620	607
SYMBOL	155	150, 151, 352, 386, 424, 488, 675, 794, 811
SYNCHRONIZATION	666	417
TAKE-STATEMENT	792	688
TASK-DECLARATION	391	188
TASK-MODE	341	210
TERMINATE-STATEMENT	640	610
TITLE	700	695, 713, 719
TRANSFER-DIRECTION	857	850, 851, 852, 853
TRANSPUT	679	419
UPON	706	696, 719

PEARL-AVIONIC-Subset

3. Cross-Reference-List of the Literals

SYMBOL	LITERALE	DEFINITION	VERWENDUNG
#			49
,			50
\$			43, 43, 57, 57, 58, 58, 59, 59
(50
			50, 145, 151, 162, 167, 172, 195,
			296, 307, 317, 323, 332, 337, 350,
			364, 384, 445, 489, 601, 701, 736,
			752, 773, 775, 800, 806, 831, 880,
			909
)			50, 145, 151, 162, 167, 172, 195,
			296, 307, 317, 323, 332, 337, 351,
			365, 385, 445, 489, 601, 701, 736,
			752, 774, 775, 800, 806, 831, 880,
			910
*			49, 521, 889, 896, 916
**			527
+			38, 49, 513, 532, 741, 917
,			51, 145, 151, 162, 195, 203, 238,
			296, 307, 317, 323, 351, 354, 365,
			385, 387, 445, 736, 752, 774, 781,
			831, 897
-			38, 49, 514, 533, 741
->			860
/			31, 31, 51, 157
//			49, 522
/=			523
0			502
1			25, 63
2			25, 63
3			25
4			25
5			26
6			26
7			26
8			26
9			20
:			20
			50, 84, 85, 187, 405, 407, 874, 902,
			910
:=			424
;			51, 172, 173, 173, 175, 176, 181,
			183, 185, 188, 368, 374, 408, 543
<			52, 506
<-			858
<->			859
<=			508
<>			515
=			52, 424
==			501

PEARL-AVIONIC-Subset

>	52,587
><	516
>=	589
A	68,78,772
ACTIVATE	616
AFTER	654
ALL	660
ALT	576
ANALOG	281
AND	497
AT	653,736
B	57,68,78,772
B1	57
B3	58,772
B4	59,772
BEGIN	543
BIT	231,538
BR	836
BY	586
C	68,78
CALL	596
CASE	575
CAT	515
CHAR	231,537
CLOCK	230
CLOSE	724
COL	767
CONTINUE	626
CREATE	694
CSHIFT	516
D	68,79,775
DCL	347,381
DECLARE	347,381
DELETE	712
DEVICE	252
DIRECT	259,291
DISABLE	673
DRAW	843
DUR	230
DURING	662
DVC	252
E	38,68,79,773
ELSE	570
ENABLE	672
END	376,591
ENTRY	317
EQ	501
EXTERNAL	281
F	68,79,773
FILE	265
FIN	570,577
FIT	528
FIX	536
FIXED	194,232
FLOAT	194,232,535
FOR	582

PEARL-AVIONIC-Subset

FORWARD	291
FRAME	836
FROM	584,729,786,793,911,818,843
G	69
GE	503
GET	729
GLOBAL	337
GOTO	563
GT	507
H	69
HERE	741
HRS	92
I	69
IDENT	325,352,386
IDENTICAL	325,352,386
IF	568
INDUCE	675
INIT	350,384
INITIAL	350,384
INTERNAL	257,281
INTERRUPT	245
INV	224,324
IRUPT	245
J	69
K	69
L	69
LE	508
LENGTH	193
LINE	763
LT	506
M	70
MAP	836
MIN	94
MODEND	176
MODULE	172
MSEC	98
N	70
NE	502
NOCYCL	302
NOSTREAM	301
NOT	534
O	70
ON	406
OPEN	718
OR	493
OUT	577
P	70
PAGE	766
PREVENT	636
PRI0	399
PRIORITY	399
PROBLEM	175
PROC	364
PUT	786
Q	70
R	70

PEARL-AVIONIC-Subset

REENTRANT	368
RELEASE	667
REPEAT	591
REQUEST	667
RESIDENT	359
RESUME	631
RETURN	601
RETURNS	332
S	71
SEC	96
SEE	818
SEMA	216
SEND	811
SHIFT	517
SIGNAL	246,296
SINK	258,286
SKIP	558,765
SOURCE	258,286
SOUSI	258,286
SPC	200
SPECIFY	200
STREAM	301
STRUCT	237
SUSPEND	621
SYSTEM	173
T	71,775
TAKE	793
TASK	342,392
TERMINATE	641
THEN	569
THRU	731,788,795,813,820,845
TITLE	701
TO	588,730,797,794,812,819,844
TRIGGER	674
U	71
UNTIL	661
UPON	707
V	71
W	71
WHEN	655
WHILE	590
X	71,764
XA	837

REFERENCES

1. PEARL-Arbeitskreis, nowadays organized as the committee for realtime programming languages in the "Gesellschaft für Mess- und Regelungstechnik des VDE".
2. Gesellschaft für Kernforschung mbH,
Projekt Prozesslenkung mit DV-Anlagen;
Postfach 3640,
7500 Karlsruhe.
3. K.H.Timmesfeld, B.Schürlein, P.Rieder, K.Pfeiffer, G.Müller, K.Kreuter, P.Holleczech, V.Haase, L.Frevert, P.Elzer, S.Eichentopf, B.Eichenauer, J.Brandes: PEARL, A proposal for a process- and experiment automation realtime language; KFK-PDV1, April 1973; Gesellschaft für Kernforschung mbH, Karlsruhe.
4. Lauber, R.: Prozessautomatisierung I. Berlin-Heidelberg-New York: Springer 1976.
5. Martin, T.: Prozessdatenverarbeitung, Berlin: Elitera 1976.
6. ASME Report: Programmieranleitung für das ASME-PEARL-SUBSET/1; January 1976.

ANNEX K

**EXTRACTS FROM THE OFFICIAL DEFINITION OF
CORAL 66**

SUMMARY

This Annex comprises extracts, some paraphrased, from the "Official Definition of Coral 66", first published in 1970 by Her Majesty's Stationery Office (HMSO). A third impression, published in 1974, included certain amendments, and it is from this edition that the material contained herein has been drawn.

Basically, the Annex is drawn from the descriptive matter in the Official Definition, but includes none of the syntax nor the Appendices. This is so that the reader can appreciate the flavour of the language without being involved in undue technicality.

Complete copies of the Official Definition can be obtained from HMSO. Advice on the semantic interpretation of the language is available from:

Computing Standards Section
Royal Signals and Radar Establishment
St Andrews Road
Great Malvern
Worcs
England.

PREFACE

Coral 66 is a general-purpose programming language based on Algol 60, with some features from Coral 64 and Jovial, and some from Fortran. It was originally designed in 1966 by I.F. Currie and M. Griffiths of the Royal Radar Establishment in response to the need for a compiler on a fixed-point computer in a control environment. In such fields of application, some debasement of high-level language ideals is acceptable if, in return, there is a worthwhile gain in speed of compilation with minimal equipment and in efficiency of object code. The need for a language which takes these requirements into account, even though it may not be fully machine-independent, is widely felt in industrial and military work. We have therefore formalized the definition of Coral 66, taking advantage of experience gained in the use of the language. Under the auspices of the Inter-Establishment Committee for Computer Applications, we have had technical advice from staff of the Royal Naval Scientific Service, the Royal Armament Research and Development Establishment, the Royal Radar Establishment, the Defence ADP Training Centre, from serving officers of all three services and from interested sections of industry, to all of whom acknowledgments are due.

The present definition is an inter-service standard for military programming, and has also been widely adopted for civil purposes in the British control and automation industry. Such civil usage is supported by RRE and by the National Computing Centre at Manchester, on behalf of the Department of Industry. The NCC has agreed to provide information services and training facilities, and enquiries about Coral 66 for industrial application should be directed to that organization.

Royal Radar Establishment
Malvern
Worcs.

June, 1974

P.M. WOODWARD
P.R. WETHERALL
B. GORMAN

CONTENTS

	Page
SUMMARY	K-2
PREFACE	K-3
1. INTRODUCTION	K-7
1.1 Special-Purpose Languages	K-7
1.2 Real Time	K-7
1.3 Implementation	K-7
2. THE CORAL 66 PROGRAM	K-7
2.1 Objects	K-8
2.2 Program	K-8
3. SCOPING	K-8
3.1 Block Structure	K-8
3.2 Clashing of Names	K-8
3.3 Globals	K-9
3.4 Labels	K-9
3.5 Restrictions Connected with Scoping	K-9
4. REFERENCE TO DATA	K-9
4.1 Numeric Types	K-9
4.2 Simple References	K-9
4.3 Array References	K-9
4.4 Packed Data	K-10
4.4.1 Table Declaration	K-10
4.4.2 Table-Element Declaration	K-10
4.4.3 Example of Table Declaration	K-11
4.4.4 Reference to Tables and Table-Elements	K-11
4.5 Storage Allocation	K-11
4.6 Presetting	K-11
4.6.1 Presetting of Simple References and Arrays	K-12
4.6.2 Presetting of Tables	K-12
4.7 Preservation of Values	K-12
4.8 Overlay Declarations	K-12
5. PLACE REFERENCES - SWITCHES	K-13
6. EXPRESSIONS	K-13
6.1 Simple Expressions	K-13
6.1.1 Primaries	K-13
6.1.2 Word-Logic	K-14
6.1.3 Evaluation of Expressions	K-15
6.2 Conditional Expressions	K-15
6.2.1 Conditions	K-15
7. STATEMENTS	K-15
7.1 Assignments	K-15
7.2 Goto Statements	K-16
7.3 Procedure Statements	K-16
7.4 Answer Statements	K-16
7.5 Code Statements	K-16
7.6 Compound Statements	K-16
7.7 Blocks	K-16
7.8 Dummy Statements	K-16
7.9 Conditional Statements	K-17
7.10 For Statements	K-17
7.10.1 For-Elements with STEP	K-17
7.10.2 For-Elements with WHILE	K-17
8. PROCEDURES	K-18
8.1 Answer Specification	K-18
8.2 Procedure Heading	K-18

AD-A044 915

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT--ETC F/6 3/1
A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CON--ETC(U)
MAY 77 G E SCHWEIZER, A A CALLAWAY, E C GANGL

UNCLASSIFIED

AGARD-AR-90

NL

6 OF
ADA
044 915



END
DATE
FILMED
11-77
DDC

	Page
8.3 Parameter Specification	K-18
8.3.1 Value Parameters	K-19
8.3.2 Data Reference Parameters	K-19
8.3.3 Place Parameters	K-19
8.3.4 Procedure Parameters	K-19
8.3.5 Non-Standard Parameter Specification	K-20
8.4 The Procedure Body	K-20
9. COMMUNICATORS	K-20
9.1 COMMON Communicators	K-20
9.2 LIBRARY Communicators	K-21
9.3 EXTERNAL Communicators	K-21
9.4 ABSOLUTE Communicators	K-21
10. NAMES AND CONSTANTS	K-21
10.1 Identifiers	K-21
10.2 Numbers	K-21
10.3 Literal Constants	K-21
10.4 Strings	K-21
11. TEXT PROCESSING	K-22
11.1 Comment	K-22
11.1.1 Comment Sentences	K-22
11.1.2 Bracketed Comment	K-22
11.1.3 END Comment	K-22
11.2 Macro Facility	K-22
11.2.1 String Replacement	K-22
11.2.2 Parameters of Macros	K-22
11.2.3 Nesting of Macros	K-22
11.2.4 Deletion and Redefinition of Macros	K-23

EXTRACTS FROM THE OFFICIAL DEFINITION OF CORAL 66

1. INTRODUCTION

It is virtually impossible to design a standard language such that programs will run with equally high efficiency in all types of computer and in any applications. Much of the design of Coral 66 reflects this difficulty. For example, the language permits the use of non-standard "code statements" for any parts of a program where it may be important to exploit particular hardware facilities. A special feature is scaled fixed-point arithmetic for use in small fixed-point machines; the floating point facilities of the language can be omitted when hardware limitations make the use of floating-point arithmetic uneconomical. Other features also may be dropped without reducing the power of the language to an unacceptably low standard.

1.1 Special-Purpose Languages

A clear distinction must be made between general-purpose languages for use by skilled programmers, and more limited languages designed to incorporate the inbuilt assumptions of specialized applications or to make direct computer access practical for the non-specialist user. Coral 66 belongs to the first category. Languages in this class are suitable for writing compilers and interpreters as well as for direct application. Special-purpose languages can therefore be implemented by means of software written in Coral 66, backed up as required with suites of specialized macros or procedures. It is largely for this reason that the facilities for using procedures have been kept as general as possible. The main differences between Coral 66 procedures and those of Algol 60 lie in the replacement of the Algol 60 dynamic "name parameter" by the more efficient "location" or reference parameter used in Fortran, and the requirement to declare recursive procedures explicitly as such, again in the interest of object-code efficiency.

1.2 Real Time

The theory and structure of programming for real-time computer applications has not yet advanced to such a point that a particular choice of language facilities is inevitable. Further, the design of real-time languages is handicapped by the lack of agreed standard software interfaces for applications programmers or compiler writers. This does not imply that real-time programs cannot yet be written in high-level language. The use of Coral 66 in real-time applications implies the presence of a supervisory system for the control of communications, which may have been designed independently of the compiler. The programmer's control over external events, and the computer's reaction to them, is expressed by the use of procedures or macros which communicate with the outside world indirectly through the agency of the supervisory software. No fixed conventions are laid down for the names or action of such calls on the supervisor.

1.3 Implementation

Considerations of software engineering have been allowed to influence the design of Coral 66, principally to ensure the possibility of rapid compilation, loading and execution. Conceptually, Coral 66 compilation is a one-pass process. The insistence that identifiers are fully declared or specified before use simplifies the compiler by ensuring that all relevant information is available when required. The syntax of the language is transformable into one-track predictive form, which enables fast syntax analysers with no back-tracking to be employed. Features which require elaborate hardware in the object machine for efficient program execution, for example dynamic storage allocation, are not included in the language. Unless run in a special diagnostic mode, a Coral 66 compiler is not expected to generate run-time checks on subscript bounds. No run-time checking of procedure entries is necessary. The arrangements for separate compilation of program segments are designed to minimize load-time overheads, but the specification of the interface between a Coral 66 compiler and the loader is outside the scope of the present document.

2. THE CORAL 66 PROGRAM

A distinction is made between *symbols* and *characters*. Characters, standing only for themselves, may be used in "strings" or as literal constants. Apart from such occurrences, a program is regarded as a sequence of symbols, each visibly representable by a unique character or combination of characters. The symbols of the language are defined, but the characters are not. For the purpose of the language definition, words in upper case letters are treated as single symbols. Lower case letters are reserved for use in *identifiers*, which may also include digits in non-leading positions. Except where they are used in strings, layout characters are ignored by a Coral 66 compiler.

2.1 Objects

A program is made up of symbols (such as BEGIN, =, 4) and arbitrary identifiers which, by declaration, specification or setting acquire the status of single symbols. Identifiers are names referring to *objects* which are classified as

data (numbers, arrays of numbers, tables)
places (labels and switches)
procedures (functions and processes).

2.2 Program

A program need not be compiled in one unit, but may be divided into *segments* for separate compilation. To make it possible to refer to chosen objects in different segments, the names and types of such objects are written outside the program segments in *communicators*. Objects fully defined within the program are rendered accessible to all segments by their mention in a COMMON communicator (Sections 3.3 and 9.1). Objects whose full definition lies outside the program, for example library procedures, can be made accessible to all segments by mention in forms of communicator whose definition will be implementation-dependent. A Coral 66 program will thus comprise

name of program
optional communicators
named segments

in some appropriate sequence. Each program segment is in the form of a *block* (Section 3). The language definition does not specify how the program or its segments shall be named or how the segments are to be separated or terminated, but when a whole program is compiled together, a typical form might be:

name of program
COMMON etc.;
segment name 1
BEGIN ... END;
segment name 2
BEGIN ... END
FINISH.

The program starts running from the beginning of a segment, the choice of which will depend upon a convention or mechanism outside the definition of the language.

3. SCOPING

A named object can be brought into existence for part of a program and may have no existence elsewhere (but see Section 4.7). The part of the program in which it is declared to exist is known as its *scope*. One effect of scoping is to increase the freedom of choosing names for objects whose scopes do not overlap. The other effect is economy of computer storage space. The scope of an object is settled by the block structure of the program as described below.

3.1 Block Structure

A block is a statement consisting, internally, of a sequence of *declarations* followed by a sequence of *statements* punctuated by semi-colons and all bracketed by a BEGIN and END.

The declarations have the purpose of fully classifying new objects and providing them with names (identifiers). As a statement can be itself a block merely by having the right form, blocks may be nested to an arbitrary depth. Except for global objects (Section 3.3), the scope of an object is the block in which it is declared, and within this block the object is said to be *local*. The scope penetrates inner blocks, where the object is said to be *non-local*.

3.2 Clashing of Names

If two objects have the same name and their scopes overlap, the clash of definitions could give rise to ambiguity. Typically, a clash occurs when an inner block is opened and a local object is declared to have the same name as a non-local object which already exists. In this situation, the non-local object continues to exist through the inner block (e.g. a variable maintains its value), but it becomes temporarily inaccessible. The local meaning of the identifier always takes precedence.

3.3 Globals

A program consists of a number of segments, each of which may be described as an *outermost block*, as there is no formal block surrounding the segments. In addition to objects which are local to inner blocks or outermost blocks, *global* objects may be defined. Such objects may be used in any segment, as their scope is the entire program. To become global, an object must be named in a communicator written outside the segments. For some types of object, such as COMMON data references, this takes the form of a declaration (and is the only declaration required). Other types of object, specifically COMMON labels, COMMON switches and COMMON procedures, must be fully defined within a segment. This means that COMMON labels must be set, and COMMON switches and procedures must be declared, in one of the outermost blocks of the program. Such objects are merely "specified" in the COMMON communicator, as described in Section 9.1, and are treated as local in every outermost block of the program. Global objects declared outside the segments are treated as non-local. All globals are non-local in all the inner blocks of any segment. With these rules of locality, questions of clashing are resolved in accordance with Section 3.2.

3.4 Labels

Any statement may be labelled by writing in front of it an identifier and a colon. The scope of a label is the smallest block embracing the statement which is labelled, extending from BEGIN to END. Thus labels can be used before they have been set. It also follows that the only means of entering an inner block is through its BEGIN. It is possible to jump into an outermost block from a different segment by the use of a COMMON label (or switch or procedure).

3.5 Restrictions Connected with Scoping

No identifier other than a label may be used before it has been declared or specified. Specification means that the type of object to which an identifier refers has been given, but not necessarily the full definition of the object (see Section 9.1). Typically, a procedure identifier is specified as referring to a certain type of procedure with certain types of parameters by the heading of the procedure declaration, but the procedure is not fully defined until the end of the declaration as a whole. As an example of this, assume that two procedures *f* and *g* are declared in succession after the beginning of a segment. Then the body of *g* may call on itself or on the procedure *f*, but the body of *f* may not call on the procedure *g* unless *g* has been specified in a COMMON communicator. If a procedure is defined in a manner which directly or indirectly calls on itself, that procedure is said to be recursive and must be explicitly declared as such.

4. REFERENCE TO DATA

4.1 Numeric Types

There are three types of number, floating point, fixed-point and integer. Except in certain part-word table-elements (Section 4.4.2.2), all three types are signed. Numeric type is indicated by the word FLOATING or INTEGER, or by the word FIXED followed by scaling constants which must be given numerically, e.g.

FIXED (13,5)

This specifies five fractional bits and a minimum of 13 bits to represent the number as a whole, including the fractional bits and a sign. The number of fractional bits may be negative, zero or positive, and may cause the binary point to fall outside the significant field of the number. It is assumed throughout this definition that a number is confined within a single computer word. If, in any implementation, a different system is adopted, e.g. two words for a floating-point number, a systematically modified interpretation of the language definition will be necessary.

4.2 Simple References

The simplest objects of data are single numbers of floating, fixed-point or integer types. Identifiers may refer to such objects if suitably declared, e.g.

INTEGER *i, j, k*;
FIXED (13,5) *x, y*;

and the declarations may optionally include assignment of initial values. This is known as presetting and is described in Section 4.6.

4.3 Array References

An array is restricted to a one or two-dimensional set of numbers all of the same type (including scale for fixed-point). An array is represented by an identifier, suitably declared with, for each dimension, a lower and upper index bound in the form of a pair of integer constants, e.g.

```

FIXED (13,5) ARRAY b[0:10];
FLOATING ARRAY c[1:3, 1:3];

```

The lower bound must never exceed the corresponding upper bound. If more than one array is required with the same numeric type, and the same dimensions and bounds, a list of array identifiers separated by commas may replace the single identifiers shown in the above examples. Arrays with the same numeric type but different bounds or dimensions may also be included in a composite declaration, as shown below.

```

INTEGER ARRAY p, q, r[1:3], s[1:4], t, u[1:2, 1:3];

```

An array identifier refers to an array in its entirety, but its use in statements is confined to the communication of the array reference to a procedure. Elsewhere, an array identifier must be indexed so that it refers to a single array element. The index, in the form of an arithmetic expression enclosed in square brackets after the array identifier, is evaluated to an integer as described in Section 6.1.3.

4.4 Packed Data

There are two systems of referring to packed data, one in which an unnamed field is selected from any computer word which holds data (see Section 6.1.1.2.2), and one in which the data format is declared in advance. In the latter system, with which this section is concerned, the format is replicated to form a *table*. A group of n words is arbitrarily partitioned into bit-fields (with no fields crossing a word boundary), and the same partitioning is applied to as many such groups (m say) as are required. The total data-space for a table is thus nm words. Each group is known as a *table-entry*. The fields are named, so that a combination of field identifier and entry index selects data from all or part of one computer word, known as a *table-element*. The elements in an entry may occupy overlapping fields, and need not together fill all the available space in the entry.

4.4.1 Table Declaration

A table declaration serves two purposes. The first is to provide the table with an identifier, and to associate this identifier with an allocation of word-storage sufficient for the width and number of entries specified. For example,

```

TABLE april[3,30]

```

is the beginning of a declaration for the table "april" with 30 entries each 3 words wide, requiring an allocation of 90 words in all. The second purpose of the declaration is to specify the structure of an entry by declaring the elements contained within it, as defined in Section 4.4.2 below. Data-packing is implementation dependent, and the examples will be found to assume a wordlength of 24 bits.

4.4.2 Table-Element Declaration

A table-element declaration associates an element name with a numeric type and with a particular field of each and every entry in the table. The field may be the whole or part of a computer word, and the form of declaration differs accordingly.

4.4.2.1 Whole-word table-elements

The form of declaration for whole-word table-elements is an identifier, followed by a number type and a word-position. For example,

```

tickets INTEGER 0

```

declares a pseudo-array of elements named "tickets". (True array elements are located consecutively in store, Section 4.5.) Each element refers to a (signed) integer occupying word-position zero in an entry. Similarly,

```

weight FIXED (16, -4) 1

```

locates "weight" in word-position 1 with a significance of 16 bits, stopping 4 bits short of the binary point. Floating-point elements are similarly permitted. Word-position and bit-position (see Section 4.4.2.2) are numbered from zero upwards, and the least significant digit of a word occupies bit-position zero. A Coral 66 compiler will assume that table elements fall within the declared width of the table.

4.4.2.2 Part-word table-elements

Elements which occupy fields narrower than a computer word (and only such elements) are declared in forms such as

```

rain UNSIGNED (4,2) 2,0;
humidity UNSIGNED (6,6) 2,8;
temperature (10,2) 2,14

```

for *fixed-point elements*. The fixed-point scaling is given in brackets (total bits and fraction bits), followed by the word- and bit-position of the field within the entry. Word-position is the word within which the field is located, and bit-position is the bit at the least significant end of the field. The word UNSIGNED increases the capacity of the field for positive numbers at the expense of eliminating negative numbers. For example, (4,2) allows numbers from -2.00 to 1.75, whilst UNSIGNED (4,2) allows them from 0.00 to 3.75. If the scale contains only a single integer, e.g.

sunshine UNSIGNED (4) 2,4;

the number in brackets represents the total number of bits for a *part-word integer*. Though (4,0) and (4) have essentially the same significance, the fact that (4,0) indicates fixed-point type and (4) indicates an integer should be borne in mind when such references are used in expressions.

4.4.3 Example of Table Declaration

```
TABLE april [3,30]
[tickets INTEGER 0;
 weight FIXED (16, -4) 1;
 rain UNSIGNED (4,2) 2,0;
 sunshine UNSIGNED (4) 2,4;
 humidity UNSIGNED (6,6) 2,8;
 temperature (10,2) 2,14];
```

It should be noted that all the numbers used to describe and locate fields must be constants. To improve program legibility, it is suggested that the word BIT be permitted as an alternative to the comma between word-position and bit-position, e.g.

sunshine UNSIGNED (4) 2 BIT 4;

4.4.4 Reference to Tables and Table-Elements

A table-element is selected by indexing its field identifier. To continue from the example in Section 4.4.3, the rain for april 6th would be written rain[5], for it should be noted that an entry always has the conventional lower bound of zero. In use, the names of table-elements are always indexed. On the other hand, a table identifier such as "april" may stand on its own when a table reference is passed to a procedure. The use of an index with a table identifier does *not* (other than accidentally) select an entry of the table. It selects a computer word from the table data regarded as a conventional array of single computer words, with lower index bound zero. Thus the implied bounds of the array "april" are 0 : 89. A word so selected is treated as a signed integer, from which it follows that april[6] in the example would be equivalent to tickets[2]. A table name is normally indexed only for the purpose of running through the table systematically, for example to set all data to zero, or to form a base for overlaying (Section 4.8).

4.5 Storage Allocation

Computer storage space for data is allocated automatically at compile time, one word for each simple reference, one for each array element, and as many as are declared for each table-entry. In any one composite declaration, a Coral 66 compiler is explicitly required to perform allocation serially. For example, the declarations

```
INTEGER a, b, c;
INTEGER p, q;
```

will make the locations of a, b, c become n, n + 1, n + 2 respectively, and those of p, q become m, m + 1 where n and m are undefined and unrelated. In two-dimensional arrays, the second index is stepped first: the declaration

```
INTEGER ARRAY a[1:2], b[1:2, 1:2];
```

will locate the elements

```
a[1], a[2], b[1,1], b[1,2], b[2,1], b[2,2]
```

in consecutive ascending locations.

4.6 Presetting

Certain objects of data may be initialized when the program is loaded into store by the inclusion of a presetting clause in the data declaration. Presetting is not dynamic, and preset values which are altered by program are not reset unless the program or segment is reloaded. An object is not eligible for presetting if it is declared anywhere within

- (a) the body of a recursive procedure, or
- (b) an inner block of the program, or
- (c) an inner block of a procedure body.

Procedure bodies do not count as blocks for the purpose of (b). For example, the integer *i* is eligible for presetting in a segment which begins as follows:

```
BEGIN PROCEDURE f ;
      BEGIN PROCEDURE g ;
            BEGIN INTEGER i ;
```

4.6.1 Presetting of Simple References and Arrays

The preset constants are listed at the end of the declaration after an assignment symbol, and are allocated in the order defined in Section 4.5. As examples,

```
INTEGER a, b, c ← 1, 2, 3 ;
INTEGER ARRAY k[1:2, 1:2] ← 11, 12, 21, 22 ;
```

If desired for legibility, round brackets may be used to group items of the presetlist, but such brackets are ignored by the compiler. The number of constants in the presetlist must not exceed, but may be less than, the number of words declared, and presetting ceases when the presetlist is exhausted. In special cases (see Section 4.7), the preset assignment symbol may be the only part of the presetlist which is present.

4.6.2 Presetting of Tables

There are two alternative mechanisms. If the internal structure of a table is completely disregarded, the table can be treated as an ordinary one-dimensional array of whole computer words (4.4.4), and preset as such (4.6.1). Alternatively, all the table-elements may be preset after their declaration list; for example,

```
TABLE gears [1,3]
[ teeth1 UNSIGNED (6) 0,0;
  teeth2 UNSIGNED (6) 0,6;
  ratio UNSIGNED (11,5) 0,12;
  arc UNSIGNED (5,5) 0,12
PRESET (57,19,3.0, ), (50,25,2.0, ), (45,30,1.5, ) ] ;
```

For table-element presetting, the word PRESET is used instead of the assignment symbol of 4.6.1. Each entry of the table is preset in succession as a group of elements, taken in the order of their declaration. Voids in the list imply absence of assignment. This may be necessary to avoid duplication when fields overlap, as do "ratio" and "arc" in the above example. As in 4.6.1, brackets used for grouping constants in the list of presets are ignored by the compiler.

The previous example could, with equal effect but less convenience, be expressed in the form

```
TABLE gears [1,3]
[ teeth1 UNSIGNED (6) 0,0;
  teeth2 UNSIGNED (6) 0,6;
  ratio UNSIGNED (11,5) 0,12;
  arc UNSIGNED (5,5) 0,12 ]
← OCTAL(1402371), OCTAL(1003162), OCTAL(603655);
```

4.7 Preservation of Values

Objects of data may have no existence outside the scope of their declarations. The values to which local identifiers refer must in general be assumed undefined when a block is first entered and whenever it is subsequently re-entered. This is due to the fact that a block-structured language is designed for automatic overlaying of data. Local working space may therefore have been used for other purposes between one entry to a block and the next. In Coral 66 this is not invariably the case. When a data declaration contains a presetlist as permitted by the rule given in Section 4.6, the values of all the objects named in that declaration will remain undisturbed between successive entries to the block or procedure body, like "own" variables in Algol 60. It is sufficient that a preset assignment symbol appears at the end of the declaration, even though the list of preset constants is void.

4.8 Overlay Declarations

Overlaying may be found desirable when COMMON data is required in some segments and not in others, as it enables global data space to be re-used for other purposes. However, indiscriminate use of overlaying should be

avoided, as it can lead to confusion and obscurity. The facility causes apparently different data references to refer simultaneously to the same objects of data, i.e. as alternative names for the same storage locations. To form an overlay declaration, an ordinary data declaration is preceded by a phrase of the form

OVERLAY Base WITH

where Base is a data reference which has previously been covered by a declaration in the same COMMON communicator or in the same segment. The base may be a simple reference, one-dimensional array reference or a table reference treated as a one-dimensional array of whole words. If the array or table identifier is not indexed, it refers to the location of its zero'th element (which may be conceptual). Storage allocated by the overlay declaration starts from the base, proceeds serially (as in 4.5) and will not be overlaid by succeeding declarations unless these are themselves overlay declarations.

5. PLACE REFERENCES – SWITCHES

Place references refer to positions of program statements, and the simplest position marker is the *label* (Section 3.4). A *switch* is a preset and unalterable array of labels, which must be within scope at the switch declaration. Any use of the indexed switch name refers to the corresponding label. For example, the switch declaration

SWITCH s ← a, b, c

causes s[1] to refer to the label a, s[2] to b and s[3] to c.

6. EXPRESSIONS

The term "expression" is reserved for *arithmetic expressions*. Coral 66 has no designational expressions of Algol 60 type. As there are no Boolean variables and no bracketed Boolean expressions (see Section 6.2.1), the expressions after IF are known as *conditions*.

6.1 Simple Expressions

Arithmetic is performed with the monadic and diadic adding operators + and −, and with the diadic multiplying operators * (multiply) and / (divide). The plus and minus operators work on *terms*, which are combinations of *factors* joined by multiplication or division. There is no exponentiation operator.

6.1.1 Primaries

Primaries are the basic operands in expressions. For example, in the analysis of the expression

$$x + y * (a + b) - 4$$

we discover three terms, the primary x, the term y*(a + b) and the primary 4. The middle term is the product of two factors, the primary y and the primary (a + b). To complete the analysis, all expressions from within brackets are similarly analysed until no further reduction is possible and no expression brackets remain. When an expression contains no word-logical operators (see Section 6.1.2), a factor must be a primary, which may or may not be of a defined type.

6.1.1.1 Untyped primaries

Untyped primaries are those operands which cannot be classed as integer, floating-point or fixed-point (of known scale) without reference to their context. For example, the number 3.1416 may be represented, with varying degrees of accuracy, in many different ways within a computer word. The same applies to an expression, whose type is determined by context (Section 6.1.3).

6.1.1.2 Typed primaries

6.1.1.2.1 Word references

A simple reference, or a reference to an array element or whole-word table-element, has a type defined in its declaration. Such references may be described as *word references* because they refer to items of data for which whole computer words are set aside. A further kind of word reference, the *anonymous reference*, takes the form

[Index]

where the index is any expression evaluated as an integer to give the actual location of a computer word. An anonymous reference possesses all the properties of an identified reference, except that it lacks an identifier. Just as a

variable *i*, declared as INTEGER *i*, may be used in an expression to refer to the contents of the computer word allocated to *i*, so the use of an anonymous reference in an expression will refer to the contents of the address defined by Index. Such contents are taken to be of numeric type INTEGER, irrespective of any declaration which may have associated that word with some other type. See also Section 6.1.1.2.3.

6.1.1.2.2 Part-words

Any single item of packed data may act as a typed primary. Such an item is either

- (a) a reference to a part-word table-element, or
- (b) a specified field of any typed primary.

In case (a), the type is defined in the table declaration. In case (b), the desired field is selected by a prefix of the form

BITS [Totalbits , Bitposition]

in front of the primary to be operated upon. The result of this operation is a positive* integer value of width Totalbits and in units of the bit at "Bitposition". The value will in general be implementation-dependent, even though the operand must be typed, as no conventions are laid down for the internal representation of floating-point or fixed-point items of data. In all cases, however, the numeric type resulting from the application of BITS is INTEGER.

6.1.1.2.3 Locations

The computer location of any word reference is obtainable by the location operator which is written in the form

LOCATION (Wordreference)

and has a value of type INTEGER. It may be noted that if *i* and *j* refer to integers, [LOCATION(*i*)] is equivalent to *i*, and LOCATION([*j*]) is equivalent to *j*. The reasoning is as follows: LOCATION(*i*) is the address of the computer word allocated to *i*. Enclosure in square brackets forms an entity equivalent to an identifier standing for this address, which by hypothesis is *i*. Similarly, [23] is equivalent to an identifier for the address 23, and LOCATION([23]) is the address for which this fictitious identifier stands, which is 23 by hypothesis.

6.1.1.2.4 Explicit type-changing

A typed primary may have its type changed, and an untyped primary may be typed, by enclosure within round brackets preceded by a specific Numbertype as described in Section 6.1.3.

6.1.1.2.5 Functions

The call of a typed procedure (Section 8) may be treated as a function and used as a primary in any expression.

6.1.1.2.6 Integers

An integer used in any expression (Section 10.2) can be assumed to have the numeric type INTEGER before any necessary type-changes are enforced by context.

6.1.2 Word-Logic

Three diadic logical operators are defined for use between typed primaries. The effect of these operators is implementation-dependent to the extent that the word-representation of data is not defined by the language. The *i*th bit of the result is a given logical function of the *i*th bits of the two operands, and the result as a whole has the numeric type INTEGER. To avoid confusion with Boolean operators in "conditions" (Section 6.2.1), a different terminology is used. The operators are

DIFFER	UNION	MASK
$\begin{array}{c cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$	$\begin{array}{c cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}$	$\begin{array}{c cc} & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$

DIFFER is recognizable as "not equivalent", UNION as "inclusive or" and MASK as "and". The operators are shown in order of increasing tightness of binding. As bracketed expressions are untyped, the use of brackets to overcome binding priorities entails explicit integer scaling. For example,

a MASK INTEGER (b UNION c)

* It is assumed that Totalbits will not be set equal to the full wordlength.

6.1.3 Evaluation of Expressions

Expressions are used in assignment statements, as value parameters of procedures and as integer indexes, all of which contexts determine the numeric type finally required. Coral 66 expressions are automatically evaluated to this type, but in the process of calculation, data may be subjected by the compiler to various intermediate transformations. Although an algorithm for evaluating expressions does not form part of the official definition of the language, all syntactically outermost terms in an expression will be evaluated to the required numeric type before the adding operators are applied. In the simplest cases, this rule ensures predictable results, though it should be particularly noted that rounding-off errors will not be minimal, and overflow may occur. If an expression is enclosed in round brackets, its terms are not "outermost", the rule no longer applies, and the algorithm for the particular compiler determines the sequence of events. The programmer can impose any desired system of evaluation by the use of Numbertype (Expression), which is a typed primary (Section 6.1.1.2), any occurrence of which behaves like a variable, ref (say), declared as

Numbertype ref;

and assigned a value by

ref \leftarrow Expression

before it is used. For example, if *i* and *j* are integer references and *x* is a floating-point reference, the assignment statement

$x \leftarrow i - j$

causes *i* and *j* to be converted to floating-point before subtraction, whilst

$x \leftarrow \text{INTEGER}(i - j)$

causes subtraction of integers before conversion to floating-point. Although the order of evaluation of an expression is undefined, the following rule concerning functions will apply. Value parameters of a function are necessarily evaluated before the function itself is computed, so that the order of evaluation of $\sin(\cos(\text{expn}))$ will be *expn*, *cos*, *sin*. Apart from this type of reversal, functions occurring in a simple expression will be evaluated in the order in which they appear when the expression is read from left to right, regardless of brackets.

6.2 Conditional Expressions

A conditional expression has the form

IF condition THEN expression ELSE expression;

with the usual interpretation.

6.2.1 Conditions

A condition is made up of arithmetic comparisons connected by Boolean operators OR and AND, of which AND is the more tightly binding. The permitted arithmetic comparisons are less than, less than or equal to, equal to, greater than or equal to, greater than, and not equal to, represented by $<$, \leq , $=$, \geq , $>$ and \neq respectively.

The Boolean operators have their usual meanings, the OR being inclusive. Conditions are evaluated from left to right only as far as is necessary to determine their truth or falsity.

7. STATEMENTS

Statements are normally executed in the order in which they are written, except that a goto statement may interrupt this sequence without return, and a conditional statement may cause certain statements to be skipped.

7.1 Assignments

The left-hand side of an assignment statement is always a data reference, and the right-hand side an expression for procuring a numerical value. The result of assignment is that the left-hand side refers to the new value until this is changed by further assignment, or until the value is lost because the reference goes out of scope (but see Section 4.7). The expression on the right-hand side is evaluated to the numeric type of the reference, with automatic scaling and rounding as necessary. The left-hand side may be a word reference as defined in Section 6.1.1.2.1 or it may be a *part-word reference*, i.e. a part-word table-element or some selected field of a word reference. When assignment is made to a part-word reference, the remaining bits of the word are unaltered. As examples of assignment,

[LOCATION(*i*) + 1] \leftarrow 3.8

has the effect of placing the integer 4 in the location succeeding that allocated to i , and

$$\text{BITS}[2,6] \ x \leftarrow 3$$

has the effect of placing the binary digits 11 in bits 7 and 6 of the word allocated to x . This last assignment statement is treated in a similar manner to an assignment which has on its left-hand side an unsigned integer table-element. The statement

$$\text{BITS}[1,23] \ [\text{LOCATION}(i) + 1] \leftarrow 1$$

would in a 24-bit machine, force the sign bit in the indicated location to "one".

There is no form of multiple assignment statement.

7.2 Goto Statements

The goto statement causes the next statement for execution to be the one having a given label. The label may be written explicitly after GOTO, or may be referenced by means of a switch whose index must lie within the range 1 to n where n is the number of labels specified in the switch declaration. See also Section 3.4 and Section 5.

7.3 Procedure Statements

A procedure identifier, followed in parentheses by a list of actual parameters (if any), is known generally as a *procedure call*. If the procedure possesses a value, it may be used as a primary in an expression, but whether it possesses a value or not, it may also stand alone as a statement. This causes

- (i) the formal parameters in the procedure declaration to be replaced by the actuals in a manner which depends on the formal parameter specifications (see Section 8.3),
- (ii) the procedure body to be executed before the statement dynamically following the procedure statement is obeyed.

7.4 Answer Statements

An answer statement is used only within a procedure body, and is the means by which a value is given to the procedure. It causes an expression to be evaluated to the *numeric type of the procedure*, followed by automatic exit from the procedure body. It takes the form ANSWER expression.

7.5 Code Statements

Any sequence of code instructions enclosed by CODE BEGIN and END may be used as a Coral 66 statement and it is recommended that code statements provide for the inclusion of nested Coral text. The form of the code is not defined; it may be the assembly code for a particular computer, or it may be at a higher level enabling available compiler features to be exploited. The code should, above all, enable the Coral programmer to exploit all available hardware facilities of the computer. For communication between code and other statements, it must be possible to use any identifier of the program within the code statement, provided such identifiers are in scope. In some implementations, a code statement may be said to possess a value. The "statement" may then be used as a primary in an expression, like a call of a typed procedure. Though not prohibited, this is not a standard feature of Coral 66, and may not be extended to other forms of statement.

7.6 Compound Statements

A compound statement is a sequence of statements grouped between the words BEGIN and END to form a single statement, for use where the syntactic structure of the language demands. Compound statements are transparent to scopes. It is therefore permitted to goto a label which is set inside a compound statement.

7.7 Blocks

See Section 3.

7.8 Dummy Statements

A dummy statement is a void whose execution has no effect. For example, a dummy statement follows the colon in

; label: END .

7.9 Conditional Statements

The two forms of conditional statement are

IF Condition THEN Consequence
IF Condition THEN Consequence ELSE Alternative

where a Consequence is a simple statement (which may be labelled), and an Alternative is any statement.

If the condition is true, the consequence is obeyed. If the condition is false and ELSE is present, the alternative is obeyed. If the condition is false and no ELSE is present, the conditional statement has no effect beyond evaluation of the condition.

7.10 For Statements

The for-statement provides a means of executing repeatedly a given statement, the "controlled statement", for different values of a chosen variable, the "control variable", which may (or may not) occur within the controlled statement. A typical form of for-statement is

FOR $i \leftarrow 1$ STEP 1 UNTIL 4,
6 STEP 2 UNTIL 10,
15 STEP 5 UNTIL 30
DO Statement.

Other forms are exemplified by

FOR $i \leftarrow 1, 2, 4, 7, 15$ DO Statement

which is self-explanatory, and

FOR $i \leftarrow i + 1$ WHILE $x < y$ DO Statement.

In the latter example, the clause " $i + 1$ WHILE $x < y$ " counts as a single for-element and could be used as one element in a list of for-elements (the "for-list"). As each for-element is exhausted, the next element in the list is taken.

The controlled variable is a word reference, i.e. with an anonymous reference or a declared word reference.

7.10.1 For-Elements with STEP

Let the element be denoted by

$e1$ STEP $e2$ UNTIL $e3$.

In contrast to Algol 60, the expressions are evaluated once only. Let their values be denoted by $v1$, $v2$ and $v3$ respectively. Then

- (i) $v1$ is assigned to the control variable,
- (ii) $v1$ is compared with $v3$. If $(v1 - v3) * v2 > 0$, then the for-element is exhausted, otherwise
- (iii) the controlled statement is executed,
- (iv) the value $v1$ is set from the controlled variable, then incremented by $v2$ and the cycle is repeated from (i).

7.10.2 For-Elements with WHILE

Let the element be denoted by

$e1$ WHILE Condition.

Then the sequence of operation is

- (i) $e1$ is evaluated and assigned to the control variable,
- (ii) the condition is tested. If false, the for-element is exhausted, otherwise
- (iii) the controlled statement is executed and the cycle repeated from (i).

Unlike those in Section 7.10.1, the expression $e1$ and those occurring in the condition are evaluated repeatedly.

8. PROCEDURES

A procedure is a body of program, written out once only, named with an identifier, and available for execution anywhere within the scope of the identifier. There are three methods of communication between a procedure and its program environment.

- The body may use formal parameters, of types specified in the heading of the procedure declaration and represented by identifiers local to the body. When the procedure is called, the formal parameters are replaced by *actual* parameters, in one-to-one correspondence.
- The body may use non-local identifiers whose scopes embrace the body. Such identifiers are also accessible outside the procedure.
- An answer statement within the procedure body may compute a single value for the procedure, making its call suitable for use as a function in an expression. A procedure which possesses a value is known as a *typed procedure*.

The form of a procedure declaration is:

```
Answerspec PROCEDURE Procedureheading;
Statement
or: Answerspec RECURSIVE Procedureheading;
Statement
```

The second of the above alternatives is the form of declaration used for recursive procedures (see Section 3.5). The statement following the procedure heading is the procedure body, which contains an answer statement (Section 7.4) unless the answer specification is void (8.1), and is treated as a block whether or not it includes any local declarations (8.4).

8.1 Answer Specification

The value of a typed procedure is given by an answer statement (Section 7.4) in its body; and its numeric type is specified at the front of the procedure declaration. An untyped procedure has no answer statement, possesses no value, and has no answer specification in front of the word PROCEDURE or RECURSIVE.

8.2 Procedure Heading

The procedure heading gives the procedure its name which is an identifier. It also describes and lists any identifiers used as formal parameters in the body. On a call of the procedure, the compiler sets up a correspondence between the actual parameters in the call and the formal parameters specified in the procedure heading.

8.3 Parameter Specification

Any object can be passed to a procedure by means of a parameter, whether it be an object of data, a place in the program, or a process to be executed. For data, there are two distinct levels of communication, *numerical values* (for input to the procedure) and *data references* (for input or output). Table I lists all the types of object which can be passed, the syntactic form of specification, and the corresponding form of the actual parameter which must be supplied in the procedure call.

TABLE I
Parameters of Procedures

<i>Object</i>	<i>Formal specification</i>	<i>Actual parameter</i>
Numerical value	VALUE Numbertype Id*	Expression
Location of data word	LOCATION Numbertype Id*	Wordreference
Name of array	Numbertype ARRAY Id*	Id
Name of table	Table †	Id
Place in program	LABEL Id*	Destination
Name of switch	SWITCH Id*	Id
Name of procedure	Procedurespec‡	Id

* Composite specification of similar parameters has Idlist in place of Id.

† See Section 8.3.2.3.

‡ See Section 8.3.4.

8.3.1 Value Parameters

The formal parameter is treated as though declared in the procedure body; upon entry to the procedure, the *actual* expression is evaluated to the type specified (including scaling if the numeric type is *FIXED*), and the value is forthwith *assigned* to the formal parameter. The formal parameter may subsequently be used for working space in the body; if the actual parameter is a variable, its value will be unaffected by assignments to the formal parameter.

8.3.2 Data Reference Parameters

Location, array and table parameters are all examples of data references. Upon entry to the procedure, these *formals* are made to refer to the same computer locations as those to which the actual parameters already refer. Operations upon such formal parameters within the procedure body are therefore operations on the actual parameters. For example, the values of the actual parameters may be altered by assignments within the procedure.

8.3.2.1 Word location parameters

The actual parameter must be a word reference, i.e. a simple data reference, an array element, an indexed table identifier, a whole-word table-element or an anonymous reference. Index expressions are evaluated upon entry to the procedure as part of the process of obtaining the location of the actual parameter. *The numeric type of the actual parameter must agree exactly with the formal specification.* Part-word references, such as table-elements are not allowed as word location parameters. An example of a procedure heading and a possible call of the same procedure is:

```
heading  f(VALUE INTEGER n; LOCATION INTEGER m)
call     f(LOCATION(u[i]), [j])
```

8.3.2.2 Array parameters

As in an array declaration, the specified numeric type applies to all the elements of the array named. The numeric type of the *actual* array name must agree with this formal specification. By indexing within the body, the procedure can refer to any element of the actual array.

8.3.2.3 Table parameters

The specification of a table parameter is identical in form to a table declaration except that presetting is not allowed. The table spec is

```
TABLE Id [ Width , Length ] [ Elementdeclist ]
```

The element declaration list need include only such fields as are used in the procedure body.

8.3.3 Place Parameters

8.3.3.1 Label parameters

The actual parameter must be a "destination", i.e. a label or a *switch element*. In the latter case, the index is evaluated once upon entry to the procedure. The actual parameter must be in scope at the call, even if it is out of scope where the formal parameter is used in the procedure body.

8.3.3.2 Switch parameters

The actual parameter is a switch identifier. By indexing within the procedure body, the procedure can refer to any of the individual labels which form the elements of the switch.

8.3.4 Procedure Parameters

Within the body of a procedure, it may be necessary to execute an unknown procedure, i.e. a procedure whose name is to be supplied as an actual parameter. The features of the unknown procedure must be formally specified in the heading of the procedure within which it is called. As an example, suppose that a procedure *g* has been declared as

```
FIXED (24,2) PROCEDURE g(VALUE INTEGER i,j; INTEGER ARRAY a); Statement
```

and further suppose that a procedure *q* has a formal parameter *f* for which it may be required to substitute *g*. A declaration of *q*, illustrating the necessary specification (italicised for clarity) might be

```
PROCEDURE q(LABEL b; FIXED (24,2) PROCEDURE f(VALUE INTEGER,  
VALUE INTEGER, INTEGER ARRAY); Statement
```


A typical call of *q* would be *q(lab,g)*. At the inner level of parameter specification, no formal identifiers are required, no composite specifications are allowed (as for *i* and *j* in *g*) and the specifications are separated by commas. To pursue the example to a deeper level of nesting, suppose that a procedure *c66* has a parameter *p* for which it may be required to substitute *q*. A declaration of *c66* might then be

```
PROCEDURE c66(PROCEDURE p(LABEL, FIXED (24,2) PROCEDURE); SWITCH s); Statement.
```

A typical call of *c66* would be *c66(q,sw)*. At the level of specification shown in italics in the latter example, no further parameter specifications are required.

8.3.5 Non-Standard Parameter Specification

The need to specify numeric type for formal value and location parameters places an undesirable constraint on the designer of input and output procedures. For such procedures it is desirable that the procedure should adapt itself to the numeric type and scale of the actual parameters. An acceptable means for achieving this is exemplarized by the following declaration of an output procedure:

```
PROCEDURE out (VALUE u:v)
```

where *u* and *v* are known as a formal pair.

At the call of the procedure, each formal pair corresponds to a single actual parameter. The first identifier is used within the procedure body, with numeric type integer, as a reference to the value of, or as the location of, the actual parameter. The compiler arranges that the second identifier passes the numeric type and scale of the actual parameter, represented in the form of an integer by some implementation-dependent convention.

In the above example, if *x* is a variable of numeric type *FIXED (24,12)*, the procedure statement *out(x)* would take account of this known scale.

8.4 The Procedure Body

For purposes of scoping, a procedure declaration may be regarded as a block at the place where it appears on the program sheet (even though this might be an illegal position). Everything except the body can be disregarded, and the formal parameters treated as though declared within the body, labels included. Identifiers which are non-local to the procedure body are those in scope at the place of the procedure declaration, subject to the restrictions given in Section 3.5. Actual parameters must, of course, be in scope at the procedure call. For example, the block:

```
BEGIN INTEGER i;
  INTEGER PROCEDURE p; ANSWER i;
  i ← 0;
  BEGIN INTEGER i;
    i ← 2;
    print(p)
  END
END;
```

has the effect of printing 0.

9. COMMUNICATORS

The segments of a program may communicate with each other through *COMMON* (Section 9.1 below), and with objects external to the program by means of communicators such as *LIBRARY*, *EXTERNAL* or *ABSOLUTE*, as defined in particular implementations.

9.1 COMMON Communicators

Global objects declared within a program (Section 3.3) are communicated to all segments through a *COMMON* communicator. This consists of a list of *COMMON* items separated by semi-colons all within round brackets following the word *COMMON*. Such items are of three kinds, corresponding to the division of objects into data, places and procedures. A *COMMON* data item is a declaration of the identifiers listed within it, exactly as in Section 4, storage being allocated as in Section 4.5, presets and overlays as in Sections 4.6 and 4.8. Communication of places and procedures takes the form of *specification*, as in the equivalent parameters of a procedure declaration (Sections 8.3.3 and 8.3.4). For each identifier specified in a *COMMON* communicator, there must correspond an appropriate declaration (or for labels a setting) in one and only one outermost block of the program.

9.2 LIBRARY Communicators

To make provision for the use of library procedures (and possibly also data references used by such procedures), programs may include **LIBRARY** communicators. These should begin with the word **LIBRARY** and be styled to conform with the rest of the language. The relative importance attached to **COMMON** and **LIBRARY** as means of inter-segment communication borders on questions of implementation which fall outside the scope of the present language definition.

9.3 EXTERNAL Communicators

It may be desirable to refer to an object external to a Coral 66 program by means of an identifier. Provided the loader permits, this may be achieved by an **EXTERNAL** communicator similar in form to a **COMMON** communicator.

9.4 ABSOLUTE Communicators

Coral 66 programs may refer to objects having absolute addresses in the computer by the use of **ABSOLUTE** communicators which associate an identifier with a specification of the "absolute" object, including its address. The form recommended is that of a **COMMON** communicator, except that each identifier to be associated with an absolute location takes the syntactic form *Id / Integer*.

10. NAMES AND CONSTANTS

10.1 Identifiers

Identifiers are used for naming objects of data, labels and switches, procedures, macros and their formal parameters. An identifier consists of an arbitrary sequence of lower case letters and digits, starting with a letter. It carries no information in its form, e.g. single-letter identifiers are not reserved for special purposes. It may be of any length, though it is permissible for compilers to disregard all but the first twelve printing characters. As layout characters are ignored, spaces may be used in identifiers without acting as terminators.

10.2 Numbers

Numerical constants appearing in other sections of this definition are of the following types:

- (a) *Constants* for presetting, optionally signed.
- (b) *Integers* and *reals* as primaries in expressions. A sign attached to a primary belongs syntactically to the expression and not to the number.
- (c) *Integers* and *signed integers* used in declarations or specifications, typically for defining fixed scales, bit-fields and array bounds.

10.3 Literal Constants

A printing character is assumed to have a unique integer representation within the computer, dependent on some hardware or software convention. The integer value may be referred to within the program by the **LITERAL** operator. For example,

LITERAL(a)

has an integer value uniquely representative of "a". The form is included within the syntax of integer (Section 10.2). The printing characters will be implementation-dependent, but it must be assumed that the set includes one 26-letter alphabet and a set of 10 digits. Layout characters are not acceptable as arguments of **LITERAL**.

10.4 Strings

A string is any succession of characters (printing or layout) enclosed in quotation marks (string quotes). Assuming that the hardware representations of the opening and closing quote symbols are distinguishable, occurrence of such marks must be properly paired within the string (but see Appendix 2). A string is classed as an unconditional expression (Section 6), and its value is its location, but it may not be used as a **LOCATION** parameter. Procedures capable of selecting individual characters from a string should be designed so that characters are represented by the same integer values as are defined for literal constants.

String = < sequence of characters with quotes matched >.

11. TEXT PROCESSING

11.1 Comment

A program may be annotated by the insertion of textual matter which is ignored by the compiler.

11.1.1 Comment Sentences

A comment sentence may be written wherever a declaration or statement can appear. It consists of the word **COMMENT** followed by text and terminated by a semi-colon. For obvious reasons, the text must not contain a semi-colon. The entire comment sentence is ignored by the compiler.

11.1.2 Bracketed Comment

Bracketed comment is any textual matter enclosed within round brackets immediately after a semi-colon of the program. The text may contain brackets provided that they are matched. Bracketed comment (including the brackets) is ignored by the compiler.

11.1.3 END Comment

Annotation may be inserted after the word **END** provided that it takes the form of an identifier only. The "identifier" is ignored by the compiler.

11.2 Macro Facility

A Coral 66 compiler embodies a macro processor, which may be regarded as a self-contained routine which processes the text of the Coral program before passing it on to the compiler proper. Its function is to enable the programmer to define and use convenient macro names, in the form of identifiers, to stand in place of cumbersome or obscure portions of text, typically code statements. Once a macro name has been defined, the processor expands it in accordance with the definition wherever it is subsequently used, until the definition is altered or cancelled (11.2.4). However, the macro processor treats comments and constant character strings (Section 10.4) as indivisible entities, and does not expand any identifiers within these entities. No character which could form part of an identifier may be written adjacent to the use of a macro name or formal parameter, as this would inhibit the recognition of such names. A macro definition may be written into the source program wherever a declaration or a statement could legally appear, and is removed from it by the action of the macro processor.

11.2.1 String Replacement

In the simplest use, a macro name stands for a definite string of characters, the macro body. For example, the (fictitious) code statement

```
CODE BEGIN 123,45,6 END
```

might be given the name "shift6". The macro definition would be written

```
DEFINE shift6 < CODE BEGIN 123,45,6 END >;
```

The expansion, or body, can be any sequence of characters in which string quotes are matched. Care must be taken to include brackets, such as **BEGIN** and **END**, as part of the macro body whenever there is the possibility that the context of the expansion may demand them.

11.2.2 Parameters of Macros

A macro may have parameters, as in the following example,

```
DEFINE shift(n) < CODE BEGIN 123,45,n END >;
```

Subsequent occurrences of **shift(6)** would be expanded to the code statement in 11.2.1. A formal parameter, such as **n** above, must be written as an identifier. An actual parameter (e.g. 6) is any string of characters in which string quotes are matched, all round and square brackets are nested and matched, and all occurrences of a comma lie between round or square brackets. This rule enables commas to be used for separating actual parameters. The number of actual parameters must be the same as the number of formals, which are also separated by commas.

11.2.3 Nesting of Macros

A macro definition may embody definitions or uses of other macros to any depth. When a macro is defined, the body is kept but not expanded. When the macro is used, it is as though the body were substituted into the program text, and it is during this substitution that any other macros encountered are processed. The use of a

macro with parameters may be regarded as introducing virtual macro definitions for the formal parameters before the macro body is substituted. Thus, to continue the example from 11.2.2, the occurrence of shift(6) is equivalent to

```
DEFINE n ≡ 6 ;  
CODE BEGIN 123,45,n END
```

followed immediately by deletion of the virtual macro *n*. Throughout the scope of the macro "shift", the formal parameter *n* may not be defined as a macro name. A formal parameter may not be used in any inner nested macro definition; neither in its body nor as a macro name nor as a formal parameter. Furthermore, no identifier in an actual parameter string, or its subsequent expansions, may be the same as any formal parameter of the calling macro.

11.2.4 Deletion and Redefinition of Macros

Macro definitions are valid from the point of definition until either the end of the program text is reached or the macro name is redefined or deleted. The scope of a macro is independent of the block structure of the program. To delete a macro, the command

```
DELETE Macroname ;
```

is used wherever a declaration or statement could appear. Alternatively, a macro name can be redefined. Macro definitions which have the same name are stacked, so that the most recent is the one which applies when the name is used. If a redefined macro is deleted, it is the most recent definition which is deleted, and the previous one is reinstated. "Recent" and "previous" refer to the sequence as processed by the macro processor.

ANNEX L

**CORAL 66 FOR SOFTWARE AND REAL-TIME
APPLICATIONS ON SMALLER COMPUTERS**

This report was reproduced here by kind permission of the Central Computer Agency of the UK Civil Service Department.

ACKNOWLEDGEMENTS

This report was produced on behalf of the Central Computer Agency of the Civil Service Department by Systems Programming Limited as part of a study into the implications of using types of programming languages. The study is part of a series dealing with programming and systems techniques and guidelines.

The authors of the report would like to thank the many organizations who have been of assistance in its production.

SUMMARY

The implications of using CORAL 66 are analysed, emphasising its use as a vehicle for software and real-time programming on smaller machines.

The advantages and disadvantages of the language are assessed in terms of statistical evidence obtained from existing systems and by experiment. Discussions with users have also provided further information.

The capabilities of the language are examined. The programmer productivity and run time efficiency to be expected when using CORAL 66 are compared with statistics obtained for similar work in Assembler level languages. The report concludes with a survey of user attitudes to the languages.

CONTENTS

	Page
ACKNOWLEDGEMENTS	L-2
SUMMARY	L-3
INTRODUCTION	L-5
1. CORAL 66 – HISTORY AND DESCRIPTION	L-5
1.1 Background History	L-5
1.2 The Language	L-5
1.3 Aims of the Language	L-5
1.4 Achievement of Aims	L-5
1.5 Structure of a CORAL 66 Program	L-5
1.6 Special Features of the Language	L-6
2. SCOPE AND INFORMATION REQUIRED	L-6
2.1 Technical Scope	L-6
2.2 Information Required	L-6
2.2.1 Functional Properties	L-7
2.2.2 Efficiency	L-7
2.2.3 Productivity	L-7
2.2.4 Users' Attitudes and Experiences	L-7
3. METHODS USED IN THE INVESTIGATION	L-7
3.1 Criteria for the Evaluation of CORAL 66	L-7
3.1.1 Learning-Related Criteria	L-7
3.1.2 Programming Criteria	L-8
3.1.3 Criteria Concerned with Run-Time Functions	L-8
3.2 Discussion and Interviews	L-8
3.2.1 Procedure	L-8
3.2.2 Obtaining the Results	L-9
3.2.3 Factors Affecting the Results: Functions	L-9
3.2.4 Factors Affecting the Results: Statistics	L-9
3.3 Experiments	L-9
3.3.1 Factors Affecting the Results	L-9
3.3.2 Description of the Experiments	L-9
3.3.3 Experiment No.1	L-9
3.3.4 Experiment No.2	L-10
4. ANALYSIS OF RESULTS	L-10
4.1 Functional Properties	L-10
4.1.1 Applications	L-10
4.1.2 General Programming	L-11
4.1.3 Data Structures	L-12
4.1.4 Macros and Code Inserts	L-12
4.1.5 Correcting and Testing	L-12
4.1.6 Compilers	L-13
4.2 Efficiency	L-13
4.3 Productivity	L-13
4.4 User Attitudes and Experiences	L-14
4.4.1 Learning and Understanding	L-14
4.4.2 Programmer Performance	L-14
5. CONCLUSIONS	L-14
6. BIBLIOGRAPHY/REFERENCES	L-15
7. TABLES OF RESULTS	L-16
APPENDIX 1 – Sources of Data Included in this Report	L-22
APPENDIX 2 – Implementation Using CORAL 66	L-23

CORAL 66 FOR SOFTWARE AND REAL-TIME APPLICATIONS ON SMALLER COMPUTERS

INTRODUCTION

The Central Computer Agency is preparing a series of guides on various aspects of computing to provide sound technical information for Government data processing managers. This report by SPL International describes an investigation into the advantages and disadvantages of using CORAL 66. It will be incorporated into a guide discussing the implications of language selection.

Sources of information references and tables of data obtained during the investigations are appended.

1. CORAL 66 – HISTORY AND DESCRIPTION

1.1 Background History

CORAL 66 was originally devised by I.F.Currie and M.Griffiths of the Royal Radar Establishment in 1966. The "Official Definition of CORAL 66" edited by P.M.Woodward, P.R.Wetherall and B.Gorman¹³ was published in 1970. It reflected some expansion of the language which had taken place as a result of experience in its use within real-time systems since 1966.

1.2 The Language

CORAL 66 is a general purpose programming language based on ALGOL 60, with some features from JOVIAL and FORTRAN.

1.3 Aims of the Language

The requirements that it was intended CORAL 66 should satisfy include:

- Improvement of compilation speeds (with minimal equipment);
- Improvement in efficiency of object code;
- As far as possible, machine independence.

Inevitably, such requirements meant discarding some of the more complex features of the languages to which it is related.

1.4 Achievement of Aims

Methods adopted to help to achieve the aims mentioned in Section 1.3 include:

- Insistence that identifiers (other than labels which are local to a block of code, see Section 1.5) are fully declared before they are used. In this way the work of the CORAL 66 compiler is reduced and conceptually, though not necessarily in practice, compilation is a single-pass process.
- Exclusion of features which require extra code at run time and/or elaborate hardware, e.g. dynamic allocation of storage, run-time checks on procedure entries.
- Inclusion of procedural and macro facilities allowing programmers to cope with external events although CORAL 66 itself is not designed to deal directly with input/output, timing or interrupts.
- The ability to insert code sections (i.e. in some representation of the target machine code), each of which is treated as a single CORAL statement, is particularly useful in this area.

1.5 Structure of a CORAL 66 Program

A CORAL 66 program may consist of a number of separately compiled segments, each of which has the form of an ALGOL 60 block, together with communicators which indicate those objects (data, places or procedures) in other segments which are to be accessible to this as well as data global to all segments. The outermost block of a segment may contain further inner blocks and procedures nested to an arbitrary depth.

1.6 Special Features of the Language

The Official Definition of CORAL 66 includes the following features:

- Data packing:
 - into tables, using the TABLE declaration
 - into part-words, using the BITS facility.
- Procedures:
 - with a value, e.g. typed procedure or function as in ALGOL 60
 - without a value
 - recursive - which, for reasons of efficiency, must be declared as such.

Any of these types of procedure may use formal parameters.

- Macros: with or without parameters. Macros are most frequently used to represent a group of code statements (e.g. I/O calls) but are also designed to stand in place of awkward sections of text.
- Code insertion: allowing the programmer to make full use of the hardware characteristics of a particular machine.
- Automatically scaled fixed point arithmetic.
- Floating point arithmetic: not always implemented, especially if it is uneconomical due to hardware limitations.
- Conditions are evaluated only as far as is necessary to determine their truth or falsehood.

2. SCOPE AND INFORMATION REQUIRED

2.1 Technical Scope

In any study of a programming language, there are a number of potential dangers. These include confusion of the language with a particular implementation, confusion of the language facilities offered with facilities available in the environment in which it is used, and confusion over what the language is intended to do.

In this study the "language" aspects of CORAL 66 are emphasised and wherever possible the effects of particular implementations are explained or removed. In the case of performance statistics, a range of results over a number of implementations is presented so that particular compiler effects may be seen.

The confusion of language and environmental facilities presents a similar problem. Features such as re-entrant code and test aids depend as much on the operating system used as on the language itself. An attempt has been made to isolate the purely "language" features by collecting data on CORAL 66 from a variety of environments.

Although the Official Definition of CORAL 66 describes it as a general purpose programming language, its use is examined in a more restricted field, the writing of software and real-time programs for small machines. Any exceptions to this are noted.

While this study sets out to analyse CORAL 66 as a language in the situations described, some attention must be given to individual implementations. For this reason compilers are discussed and comments about compiler requirements and limitations, as made by users, are recorded. No attempt is made to compare a number of existing compilers.

2.2 Information Required

The potential user of CORAL 66 requires more than an analysis of the language functions; he needs to be able to evaluate the cost effectiveness of using the language. The necessary data are derived from statistics on the efficiency of CORAL 66 generated code and programmer productivity rates that have been achieved.

The efficiency of CORAL 66 obviously depends on the implementation. Using a range of efficiency figures obtained from a number of different implementations, this report shows what is possible when using the various CORAL 66 compilers. Any new implementation could be expected to fall within the given range or to show an improvement.

Evaluation of programmer productivity using CORAL 66 presents a complex problem since the nature of the work, local conditions, individual skills and many other factors affect work rates. However, a range of productivity figures over a significant amount of effort in CORAL 66 has been compared with a similar range using Assembler.

Note that CORAL 66 is being compared only with Assembler or machine code. In the small machine environment in which CORAL 66 is used, this is generally the choice available to the user.

2.2.1 Functional Properties

In practice the features that a CORAL 66 system offers to the user vary with different compilers and operating systems. Areas of interest include:

- Data structures, character handling;
- Macros, re-entrant code, recursion;
- I/O and file accessing facilities;
- Diagnostic aids;
- Interfacing with other languages.

2.2.2 Efficiency

The aspect of efficiency which is of concern here is the amount of core memory occupied by the object version of a program written in the language. Such figures must be related to some other measurement in order for them to be meaningful. Statistics from programs written in Assembler give a basis for comparison. Since few programs are produced both in Assembler and in CORAL 66 some simple experiments were carried out to provide detailed information and supplement the statistics gathered.

2.2.3 Productivity

Productivity was taken to be the time from the start of coding to the point at which the program is tested and ready for use. The necessary volume of statistical information was obtained from the field. Productivity measurements for similar activities in Assembler provide a basis for comparison.

No attempt was made to derive productivity figures for the experiments referred to in 2.2.2.

2.2.4 Users' Attitudes and Experiences

Although this report is primarily concerned with the functional properties of CORAL 66, and with statistical information on its use, there are a number of areas in which informed user opinion is valuable. Typical are training and learning problems, the legibility and documentation properties of the language, the quality of staff required and the general effect on morale. Although this information is not always quantifiable, it can have significant effects on the running of an installation, and may directly affect other quantifiable properties.

3. METHODS USED IN THE INVESTIGATION

The required information was obtained by examining available literature in CORAL 66, and by interviewing a number of users. The discussions resulted in a body of opinion on the language, comments on its functional properties and a volume of statistical information.

To simplify the investigation, a set of formal criteria was drawn up to define the properties required in a language intended for use in the selected context. The criteria assisted in framing questions to be put to users and ensured consistency in the interviews.

3.1 Criteria for the Evaluation of CORAL 66

The criteria listed below were designed to cover the important areas of real-time and software situations. In some cases examples of key questions are given to clarify the extent of a criterion.

3.1.1 Learning-Related Criteria

<i>Criterion</i>	<i>Key Questions</i>
Effect on coding methods	Could standards be easily applied?
Ease of reading	-
Quality of staff required	-
Effect on morale	Do programmers like the language?
Ease of avoiding pitfalls	Are problems, perhaps leading to inefficiency, easily avoided?
Time required for productivity and proficiency	-
Cost of training	-

3.1.2 Programming Criteria

<i>Criterion</i>	<i>Key Questions</i>
I/O facilities	How does the program communicate with I/O drivers?
File access/data base	What kind of file access methods are possible? What recovery features are provided?
Ease of interfacing with other languages	Can subroutines or modules in other languages be incorporated easily?
Character handling	What character and part-word handling facilities are provided?
Data structures	Are any standard data structures provided?
Ease of program amendment and maintenance	Is modular code easy to write? Are link editors provided?
Macro facilities	Can machine code be easily used?
Extensibility	Is it possible to add to facilities (in a particular environment)?
Re-entrant code	—
Recursion	What overheads are involved?
Compile-time diagnostic aids	—
Run-time diagnostic aids	Is conditional code generation possible?
Portability	Is it easy to move a program from one machine to another? How much time and effort are required?
Production rate	—
Coding time	—
Testing time	—
Error rates	—
Reliability of product	What failure rate would be expected in programs?

3.1.3 Criteria Concerned with Run-Time Functions

<i>Criterion</i>	<i>Key Questions</i>
Efficiency of object code	—
Compiler efficiency	What demands do the various compilers make on resources for their own execution?

3.2 Discussion and Interviews

The people interviewed ranged through compiler writers and programmers writing difficult basic software, to scientists using CORAL 66 simply as a problem solving tool. A complete list of sources is given in Appendix 1.

3.2.1 Procedure

The users were asked to discuss the nature of their work, so that any statistics collected from them could be evaluated in the correct context. The functions of the language were discussed and criticisms and opinions noted. Statistics concerning productivity or efficiency were collected and references to further sources of information were recorded.

3.2.2 *Obtaining the Results*

Reports on the interviews were analysed to extract criticisms and comments concerning the language and its functions. This information was edited and classified and is analysed in Sections 4.1 and 4.4. The numerical and statistical data obtained were extracted from each report and tabulated. An analysis of the results appears in Sections 4.2 and 4.3.

3.2.3 *Factors Affecting the Results: Functions*

Certain facts should be borne in mind when considering the analysis of the results.

- The specific implementation concerned has a considerable effect on the user's attitudes and criticisms. While general opinion was consistent, special functions were praised in some places and condemned in others, largely because of a particular compiler.
- The type of work being done caused emphasis to be placed in different areas. This does not mean contradictory opinion; rather, that functions were stressed or not stressed.

3.2.4 *Factors Affecting the Results: Statistics*

When examining the statistical information gathered, the reader should consider the following factors:

- Many of the users interviewed are from research and development establishments and are not directly concerned with productivity. Individuals at comparatively few places therefore attempted to collect "productivity" statistics, and use them for planning. The productivity statistics presented were obtained by considering individual programs and systems and recording the effort involved in producing them. In view of the volume of information collected (121K of different types of program, over 9 man years of effort and five types of machine), the results are considered relevant. These productivity figures should be considered by comparison with Assembler, although the absolute figures themselves are interesting.
- Type of machine has been included in the tables. In some cases the characteristics of machine type are significant to the results.

3.3 *Experiments*

The experiments were designed to compare the object memory and core used by two implementations of the same program, one written in CORAL 66, the other in Assembler. The two programs should carry out identical tasks but it was not intended that one should merely be a transcript of the other. Each program made some attempt to exploit the facilities offered by the language in which it was written.

3.3.1 *Factors Affecting the Results*

The following factors should be borne in mind when considering the results of the experiments:

- Two different machines were used and therefore two different compilers and two Assemblers. However, the CORAL 66 and Assembler versions of the same program were always developed on the same machine.
- The CORAL 66 and Assembler versions of the same program were not always written by the same programmer.
- Even when the two versions of the same program were produced by the same programmer, the relative efficiency may be affected by his experience and proficiency in each of the two languages.

3.3.2 *Description of the Experiments*

The programs examined fall into two categories:

- A series of six text programs written for an ICL 1900 machine (Experiment 1).
- A single program which constitutes the central, processing section of a multiple-user interactive system written for a MOD1 computer with a minimum of 12K of core (Experiment 2).

3.3.3 *Experiment No.1*

Six test problems which had initially been designed for training purposes were used. They were of two types; four of them involved calculations (e.g. generation of squares, cubes or square roots, or estimation of daily supply requirements) and the remaining two were number sorts. All resulted in some printed output. Each was written in both CORAL 66 and Assembler, although not necessarily by the same programmer. No express attempt was made to optimise core usage. (It should perhaps be borne in mind when considering the results of this experiment that the largest of the programs had originally been written first in ALGOL, then written in CORAL 66 using identical logic. Alterations to the CORAL 66 version could result in a further saving in space.)

The tested version of each program was then examined and two measurements made:

- The total size of the program in the form in which it was run; i.e. including all the subroutines that it used, any extra workspace required in addition to the program work area and allowing for the fact that core was allocated to the nearest 64 word block.
- The size of the basic program plus its own data and work areas.

Results of these calculations appear in Section 7, Tables 7 and 8.

3.3.4 Experiment No.2

The second experiment examined was a program designed for a multi-user environment which involved validation and interpretation of user input, some fairly complex calculation, and the presentation of results or diagnostic codes in a readily-understandable, printed form. The program was first written in CORAL 66 then re-written in Assembler. The Assembler version is currently operational.

Unfortunately, it has not been possible to test run the CORAL 66 version as the compiler in use does not include floating-point arithmetic facilities, which are required by the program. However, this version has been very carefully checked to ensure that it is capable of doing everything that the Assembler version does.

Both the CORAL 66 and the Assembler versions were written by the same programmer.

The following measurements were made:

- Program size: in this case excluding any external routines, or work areas (see Section 7, Table 9). Since floating-point routines had not been implemented in CORAL 66 the inclusion of external routines would not provide meaningful comparisons.
- Size of each of the component parts of the program. For this measurement the program was broken down into a main program area and internal subroutines and comparative figures calculated for the two versions (see Section 7, Table 10).

4. ANALYSIS OF RESULTS

The results of the investigation are now given under four headings introduced in Section 2: functional properties, efficiency, productivity and user attitudes. Although these divisions are maintained for clarity, it will be appreciated that some comments received do not fall clearly into any one of the sections.

4.1 Functional Properties

In general CORAL 66 users were satisfied with the language and in many cases were enthusiastic about it. Some particular comments and criticisms are given below. An interesting and detailed critique of the language is provided by Pyle et al.¹⁰.

4.1.1 Applications

CORAL 66 is primarily intended as a language for writing control type real-time systems and basic software, using small machines with store in the range 16K to 64K words. The range of uses encountered conforms largely to this general class (see Appendix 2).

When used with the "Blandford Extension"⁶ it was claimed that CORAL 66 is suitable for general data processing and could be a serious rival to more widely accepted DP languages such as COBOL. The Blandford Extension itself offers character handling facilities. It is discussed further in Section 4.1.3. While CORAL 66 is generally preferred to Assembler for systems programming, the absence of character handling facilities (apart from the Blandford Extension), caused doubt as to its suitability in situations involving extensive character manipulation. However, it may be argued that CORAL 66 was primarily intended for machines without character handling capabilities, other than masking operations. Also that the part-word facilities provided in TABLE and BITS functions together with MASK, DIFFER and UNION are adequate for the control duties (such as weapons guidance and radar control) that systems would be expected to perform.

The absence of "Real-Time" features in the language was raised as a problem. However, this problem can be removed by leaving communication with the hardware and operating system as code inserts, using CODE, and the macro facility DEFINE. The language has to be viewed as a code generator, with real-time features being functions of the operating system.

An interesting comment - recorded more than once - was that CORAL 66, like ALGOL, is suitable as a "high-level flow-charting" vehicle. The implementation proceeds by defining the blocks. This is developed by Blumfield and Carmichael¹.

4.1.2 General Programming

Programmer attitudes to CORAL 66 are discussed in 4.4. This section is concerned with comments and criticisms on the structure of programs, their relationships to the system and a number of points concerning arithmetic capabilities, expressions, etc.

This whole area is particularly sensitive to the implementation of the language. Some of the comments received are more directly concerned with a particular compiler than with the language definition itself.

4.1.2.1 Program definition

The Official Definition is criticised by Pyle et al.¹⁰ for not defining clearly the concepts of program and segment. They state that it is not clear whether program means source program or object program. Much of this seems to stem from the general confusion of terminology. RRE themselves define a CORAL program as a named collection of named segments. The implementations examined resolve this confusion satisfactorily in the sense that the word program is a clearly defined term, and that entities called programs run on computers. All have named source entities (segments in the Official Definition) which are compiled to give intermediate forms ("semi-compiled", "relocatable binary", etc.). The latter are then linked together, satisfying cross-references, by some form of link editor to give absolute (object) programs, which may be loaded into a computer and run.

Unfortunately, terminology problems are introduced. Three meanings of the term segment were encountered and also the term "module", which is analogous to segment in the Official Definition. However, in any one implementation, the problem has usually been resolved adequately. Programmers are given enough information in technical documentation to enable them to work without difficulty.

4.1.2.2 Segment communication

The description of communicators is somewhat confusing. (Official Definition, 9, COMMON, LIBRARY, EXTERNAL and ABSOLUTE.) This has been variously implemented, with some of these facilities being omitted in some implementations. Included in the definition seem to be

- references to objects defined in other segments. The object would have to be declared in a segment, and referred to in another segment.
- declaration of objects which may be referred to in other segments. This means that a global entity would be declared and made available. This is the COMPOOL.

Users commented that the discipline imposed by the language in defining communication eased modular programming problems, as the number of possibilities offered by Assembler lead to dangerous situations.

4.1.2.3 Re-entrant and recursive code

Unlike ALGOL, CORAL 66 does not assume recursion; recursive procedures must be declared as such (RECURSIVE, Official Definition, 8). In the CORAL 66 Official Definition (Appendix 3) recursive procedures are defined as the highest level of implementation. They were not defined in all the compilers encountered. By not assuming recursion, dynamic stack allocation problems are reduced.

Re-entrant code is not generated by all the implementations examined. Overall re-entrancy depends on the separation of data areas and code areas, the provision of some means of reference to the data area and generated object code which does not involve any in-line coding such as return jumps or transient work areas.

To a large extent these features depend on the architecture of the machine and the operating system. For example, multiple "windows" in memory (Modular 1 segments, typically) can assist in producing re-entrant programs by having one code area and multiple data areas. An operating system which controlled its programs in a suitable manner would then be required for scheduling new activities in the programs. Within programs, a degree of re-entrancy may be obtained by having specific tasks coded as procedures. These procedures may then be called, using tables or arrays as parameters. The procedure declaration would enable the procedure body to know the structure of the table without reserving space. When the procedure is called, the actual parameter would then pass to the procedure body the work area in the form of a table or array. If the compiler stores the base of the table in a register, a re-entrant structure is obtained.

4.1.2.4 Overlays

Pyle et al. comment that no overlay facility of the OS/360 type is described in the Official Definition (this is not the CORAL 66 "OVERLAY"). The whole program has to reside in memory at the same time. However, this is surely more a concern of link editing. CORAL 66 provides the ability to write a program as a number of segments which may be compiled separately and linked together to form one absolute program. A suitable editor could be provided to map segments into various areas. Of course, the work-space used within a segment must remain within that segment space. It may not, for example, be gathered at the front of the program.

The implementations encountered did not offer this facility, although it was felt that its provision would not cause too many problems. However, in many of the systems for which CORAL 66 is used, the absence of such a feature is not critical, since backing store is not provided. For example, most guidance and control systems are held entirely in core.

An alternative solution, of course, is to provide in the operating system a means of linking independent programs, using operating system functions. This is independent of CORAL 66.

4.1.2.5 Arithmetic

CORAL 66 offers integer, fixed-point and floating-point arithmetic. Standard CORAL 66 (see Official Definition, Appendix 3) does not include floating point which depends in some cases on the provision of suitable hardware for efficiency reasons.

Some criticism was offered on fixed-point arithmetic. It was decided that fixed fractional arithmetic would be more practical, as there are a number of scaling problems in using fixed point.

4.1.3 Data Structures

CORAL 66 offers, apart from the normal numeric types INTEGER, FIXED and FLOATING, two main methods of data storage. These are TABLE and ARRAY. The Blandford Extension has expanded this beyond the Official Definition to include RECORD and BYTE ARRAY.

The data structures in the Official Definition were thought by some users to be somewhat restricted and rigid, particularly if the application involved character handling. Users in control and executive system situations generally found the TABLE structure to be adequate. Particular comments on TABLES were:

- Part-word handling may be inefficient. In some machines, this is unavoidable and users should try to avoid using part words.
- For development purposes, it would be useful if TABLE elements could cross word boundaries, even if this were inefficient.
- Arrays cannot be used as part of TABLES.

The Blandford Extension provides character handling facilities using BYTE ARRAY and RECORD. The record structure, which is somewhat similar to COBOL, permits operations on fields which consist of character strings. Field operations deliver the sequence of characters, rather than the pointer in a numerical context. It is possible to pre-set records, make "string assignments" to fields, compare fields, convert to numbers and vice versa and use records as procedure parameters. The Blandford Extension provides facilities for a useful excursion into the world of data processing. Its users are convinced of the power and value of CORAL 66 with this extension.

It was also suggested that some form of list structure would be useful, e.g. the accessing of chained blocks.

4.1.4 Macros and Code Inserts

Access to machine dependent features and all communication with the operating system is provided by the insertion of machine code using CODE or macros using DEFINE (see Official Definition 7.5 and 11.2.1). These features were thought to be very useful in critical situations, either for efficiency reasons or for direct communication with hardware, for example, I/O functions. However, such inserts should be kept to a minimum to avoid problems in portability and security. Indeed some users raised the security aspect as a particular problem.

The use of DEFINE, with macros was thought to provide access to hardware in a slightly more controlled manner. One good approach would be to provide a library of standard procedures for all system interfacing. The general user would then not require any code inserts; he would simply use library procedures.

An example of system primitives is given by Jackson and Wetherall⁹.

In general, the provision of code inserts is a pragmatic recognition of reality. Many aspects, for example "real-time" features, are properties not of a language but of an operating system. This is particularly true of the synchronising primitives used in parallel processing. To introduce these into the language definition, while in itself trivial, may lead to problems later. They are best left as macros.

4.1.5 Correcting and Testing

Two aspects must be considered here, the general ease of diagnosing and correcting errors in the language; and what specific assistance the language and its implementations offer.

4.1.5.1 Ease of fault finding

Most users find CORAL 66 programs easy to amend as trivial errors are avoided. The programmer is free to concentrate on the problem. This was thought to be particularly valuable in handling packed data in TABLES, as the possibility of error in Assembler masking and shifting is considerable. Substantial diagnostic progress is made by many users simply by working at source level and comparing actual with expected output.

4.1.5.2 Test aids provided

There was considerable criticism of the test aids provided in the language. The Official Definition provides little help. An interesting critique and some suggestions appear in Pyle et al. page 22 and seq.

Particular points raised which affect program testing were:

- A trace facility is needed. Some implementations provided this.
- Optional code generation would be useful.
- A "diagnostic mode" would be very helpful. This would permit users to introduce aids at various levels of testing. For example, it may be that a production version would require less checks than a test version of a program. The diagnostic mode could be used to control this.
- Type checking of variables in an expression could be done by a compiler.
- No index or subscript checking is done.

However, the macro facility can be used for inserting and removing diagnostic aids.

4.1.6 Compilers

As far as is possible, the effects of specific implementations have been removed from this analysis. For this reason no criticisms of particular compilers are included. However, some general comments about compilers were noted. A particular point was that compilers should be modular in construction, so that users could include only those features actually required. This was thought to be especially desirable in a development environment with facilities such as testing aids, floating point and so on, though the Official Definition could give a little more help to compiler writers. However, this suggestion was not adopted because of the added compiler complications caused by this modular approach.

4.2 Efficiency

Eight individual programs and program groups were found which had been written in both CORAL 66 and Assembler. Together they totalled approximately 67K (object size) of CORAL 66 code, and included implementations on six different machines.

In each case the size of the CORAL 66 object program was calculated as a percentage of the Assembler object program (see Section 7, Table 1). The resulting figures showed that the CORAL 66 object size ranged from 100% to 155% of the Assembler object size. In the majority of cases the figures centre around 125%–130%.

Some further comparisons were obtained of the relative object size of programs written in CORAL 66 and other high-level languages. There is not enough data in this area for any conclusions to be drawn but the figures have been appended for information (see Section 7, Table 2).

In general, timing considerations were considered less important by users than memory utilisation, as long as the order of timing change was the same as the space occupied. In the programs examined and in discussions with users, the comparison was made of execution times and space occupied when using CORAL 66 and other high-level languages. CORAL 66 was more efficient in both respects but the greater improvement was in execution time. Only rarely was it necessary to use Assembler for efficiency reasons.

4.3 Productivity

Productivity figures were collected for about 120K of programs on five machine types. The figures obtained are compared with the productivity which might be expected in similar environments using Assembler. A wide range of system types is included in the results.

In each case a record was made of the size of the program in generated object code statements, the effort in man months put into its production and the machine involved. The type of program involved was also noted. This enabled some assessment of the difficulty of the task to be made. From the collected results, the productivity in thousands of words of object code per man month was calculated. This result is given in Section 7, Table 3. For clarity of display the results are shown as a histogram. This appears in Table 4. The effort involved is measured from the beginning of design by the programmer through to delivered product.

The figures in terms of simple, medium and difficult tasks were analysed. For each type of task an approximate upper bound of productivity is given. A similar range is given for equivalent programming in Assembler. This permits the comparison of relative productivity in Assembler and CORAL 66. These figures and the sources of the Assembler programming data are given in Section 7, Table 5.

In addition some specific productivity figures as used for estimating projects by a number of manufacturers and users were obtained. These figures are given in Section 7, Table 6. They provide confirmation of the other Assembler statistics quoted.

No attempt is made to define any "absolute" productivity rates. It is worth noting that the CORAL 66 range is 0.45 to 7.3K (with the bulk of CORAL 66 results showing up to 1.75K) per man month. The results show a concentration around 0.8 to 1K per man month for "medium" to "medium difficult" tasks.

Compared with the Assembler results, the CORAL 66 figures show a sharp increase in productivity. An exact numerical comparison is difficult because the three basic ranges overlap. For example, an executive and operating system regarded as difficult was produced at a higher rate than a steering tape edit, which was medium. In general, a productivity increase by a factor of 3 seems reasonable.

Specific comments on individual items in the tables are included in the notes appended.

4.4 User Attitudes and Experiences

Comments on the use of CORAL 66 tended to fall into two major categories; remarks concerning ease of learning and understanding the language and comments on programmer performance.

4.4.1 Learning and Understanding

There was strong feeling that CORAL 66 is easy to learn, particularly for a programmer already familiar with ALGOL but rather more difficult for those who are familiar with only FORTRAN or COBOL. Some difficulty had been experienced with programmers moving from Assembler or machine code. CORAL 66 was considered to be much easier for a beginner to learn than Assembler. Although some doubts were expressed as to the lack of teaching material, References 2, 3 and 12 are examples of literature available.

There was general agreement that the language is easy to read and write. It was considered faster to write than Assembler, although the comment was made that it is also easy to write inefficient code. Productivity in the language was regarded as high.

Several users commented that since the language was easy to read, modification of programs was simplified and so time could be saved. However, CORAL 66 was not considered to be a completely self-documenting language and it was suggested that the use of long (and presumably self-defining) identifiers and clear comments was necessary to improve readability.

There was also a suggestion that a good set of rules and standards was needed to help avoid mis-use of the language. In particular, rules for the employment of machine language inserts were required to prevent unnecessary low-level coding.

4.4.2 Programmer Performance

Several users had noted improved production rates when programmers used CORAL 66. Fewer errors were made than in other languages, especially Assembler. The comment was made that improved performance could not be attributed to sophisticated test aids but rested entirely on the merits of the language. Programmers became productive sooner in CORAL 66 than in Assembler or even another high-level language. They tend to become effective quickly because they are able to grasp the whole language due to its straightforward structure.

5. CONCLUSIONS

The statistics collected show a productivity rate in CORAL 66 varying from 0.45K to 1.75K per man month, exceptionally rising to 7.3K per man month. The range of productivity which may be expected when using Assembler varies from 0.13K to 0.5K per man month. All figures are expressed in object code generated. Overall, this indicates that productivity may be increased by as much as three times if CORAL 66 is used instead of Assembler.

The efficiency measurements suggest a CORAL 66 to Assembler ratio of about 120-130% code generated, calculated as shown in Section 7, Table 1. These figures compare favourably with the relative efficiencies of Assembler and various other high-level languages as described by Henriksen and Merwin¹⁴. The comparison of execution time shows that CORAL 66 is more efficient than other high-level languages.

7. TABLES OF RESULTS

Table 1: Efficiency

The table below contains details of programs which have been written both in CORAL 66 and Assembler. Sizes are expressed in the units applicable to the machine on which the program was developed (e.g. MOD1 - 16 bit word).

<i>Program type</i>	<i>Assembler object size</i>	<i>CORAL object size</i>	<i>Machine</i>	<i>CORAL object as % of Assembler object</i>
Utility software	1380	1684	MOD 1	122%
Multiple-user interactive	1017 (d)	1286	MOD 1	126%
Radar scan driver	4750 (a)	6005	ELLIOTT 900	126%
Compiler (CORAL 66)	22000	30000	FM 1600	136%
Six test programs (sorts and calculations)	5632 (d)	7936	1900	141%
Compiler (CORAL 66)	18600	18700 (b)	1900	100%
Radio Station diagnostics	450 (c)	700	ARGUS 500	155%
System software	470	520	MYRIAD	111%

Notes: (a) The total assembler object size of this program was 5550. However a report on the development of the program states: "... there were some gross inefficiencies in the SIR program, notably use of core store to hold an unnecessary array. It would be possible by modifying the SIR program to effect a saving of about 800 words".

(b) It should be noted that some of the CORAL 66 version of the compiler has been left in PLAN code. In addition, some re-design was probably carried out to improve efficiency.

(c) This program was written in machine code.

(d) These figures were obtained by experiment. See Tables 8-10 for a detailed breakdown of the results.

Table 2: Efficiency - Further Comparisons

The following programs were written in more than one high-level language. Sizes are expressed in the units applicable to the machine on which the program was developed.

<i>Program type</i>	<i>CORAL 66 object size</i>	<i>ALGOL object size</i>	<i>COBOL object size</i>	<i>Machine</i>
Parser	6500	15500	-	1900
Test program (calculation of supplies)	2048	7360	2624	1900

Table 3: Productivity

The table contains details of programs written in CORAL 66. Program types vary from basic DP work to operating systems.

<i>Program</i>	<i>Size (K)</i>	<i>Effort (Man months)</i>	<i>Productivity (K/man month)</i>	<i>Machine</i>
Graphics software	2.0	2.5	0.8	MOD 1
Compiler (CORAL 66)	12.5	13.0	0.96	MOD 1
Basic software	3.0	3.0	1.0	MOD 1
Macro generator/link editor	4.5	3.0	1.5	MOD 1
Executive interface	5.0	3.0	1.6	MOD 1
Disc edit (a)	3.0	0.5	6.0	MOD 1
Complex Application	3.0	5.0	0.6	MYRIAD
Interactive debug aid	2.0	3.0	0.7	MYRIAD
Steering tape edit	2.0	3.0	0.7	MYRIAD
Exec/operating system	10.0	12.0	0.83	MYRIAD
Filing system	5.5	6.0	0.92	MYRIAD
On-line heat-exchange monitoring (b)	24.0	7.0	3.4	MYRIAD
Fire trajectory calculation	2.5	6.0	0.45	ELLIOTT 920
Syntax analysis	9.0	6.0	1.5	ELLIOTT 920
Radar scan control	6.0	4.0	1.5	ELLIOTT 900
Radio station diagnostics	9.0	20.0	0.45	ARGUS 500
Basic DP work (c)	11.0	1.5	7.3	1900
Compiler (CORAL 66)	7.0	4.0	1.75	1900

Notes: (a) This figure is exceptionally high and may not be very reliable. It is claimed to be correct.

(b) This task, although extensive, was regarded as easy since the programmer is highly skilled in the technical background. The entire credit for this productivity was attributed to the use of CORAL 66 for the task.

(c) Straightforward DP. Even for this the figure seems somewhat high.

General Notes

A quick and "dirty" program written in a hurry as a one-off exercise will generally be excessively large. It is reasonable to assume that this will give rise to exceptional productivity rates. That such an approach is possible using CORAL 66 will be considered by many to be an advantage of the language.

Table 4: Productivity as a Histogram

Productivity - K per man month

This chart does not include the figures of 6.0 and 7.3K per man month recorded for DP work and 3.4K per man month recorded for a calculations program.

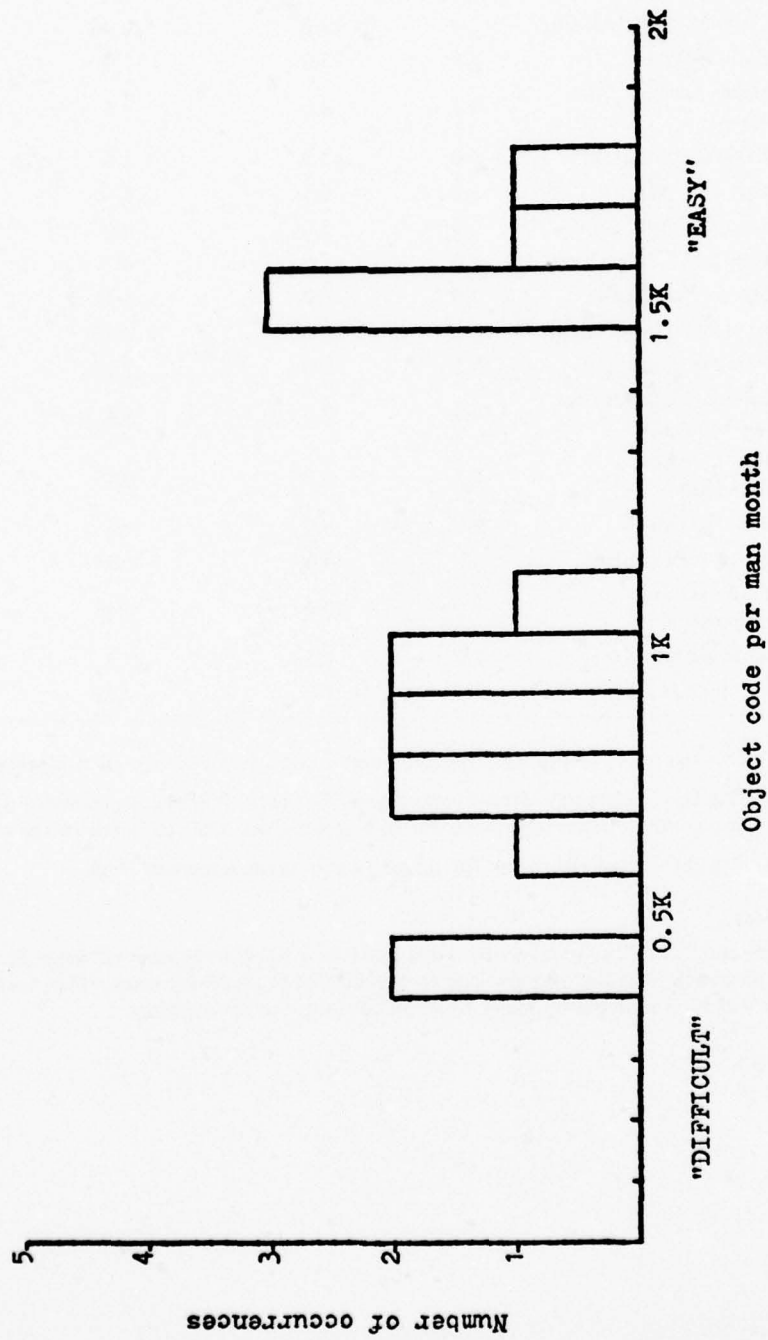


Table 5: Programmer Productivity (measured in deliverable instructions)**A. Assembler**

1. Overall range: 0.04 to 0.5K per man month.
- *2. Separated out according to difficulty of task:
 - Hard: up to 0.125K per m/m
 - Medium: up to 0.25K per m/m
 - Easy: up to 0.5K per m/m.

B. CORAL 66

1. Overall range: 0.45 to 7.3K per man month.
- *2. Separated out according to difficulty of task:
 - Hard: up to 0.83K per m/m
 - Medium: up to 1.75K per m/m
 - Easy: up to 7.3K per m/m.

*** Definitions**

- Difficult: monitors/operating systems.
- Medium: utilities/language compilers/schedulers/I/O packages.
- Easy: problem/application programs.

Note that these ranges overlap.

References for Assembler productivity figures:

- (i) Infotech state of the Art Report No.2 "Estimating Resources for large programming systems". J.D.Aron and R.W.Arthur.
- (ii) Infotech state of the Art Report No.3 "Switching Systems". Past, Present and Future. Dr M.T.Hills, University of Essex.
- (iii) Computer Management Vol.5, No.1, 15 January 1970. Commonsense about Programmer Productivity. P.A.Mimms.

Table 6: Further Information on Programmer Productivity in Assembler

In addition to figures extracted from published sources, interviews with users provided information on the range of productivity expected in Assembler programs. These may be taken into consideration when making comparisons with figures for productivity in CORAL 66. No figures were obtained for program types falling in the "easy" range.

1. Overall range (Extremely difficult to medium): 0.017 to 0.54K per man month.
2. Separated out according to difficulty of task:
 - Hard: up to 0.125K per man month
 - Medium: up to 0.54K per man month.

Table 7: Experiment Results

Experiment No.1 – Six test problems written for an ICL machine.

Comparative sizes of each test program after collection (in every case the program was successfully tested):
These figures reflect the size of program actually loaded into memory.

<i>Problem number</i>	<i>PLAN object size</i>	<i>CORAL 66 object size</i>	<i>CORAL 66 object as % of PLAN object</i>
1. Average mileage calculation	768	1152	150%
2. Three-number sort	832	1152	138%
3. Generation of numbers with their squares and cubes	704	1152	164%
4. N-number sort	896	1280	143%
5. Square roots	1024	1152	113%
6. Calculation of supplies	1408	2048	145%
TOTAL	5632	7936	141%

Table 8: Experiment No.1 (continued)

Comparative sizes of each test program before collection (i.e. actual program area with its own data area, excluding sub-routines and any extra work areas added at collection time):

<i>Problem number</i>	<i>PLAN object size</i>	<i>CORAL 66 object size</i>	<i>CORAL 66 object as % of PLAN object</i>
1.	43	101	235%
2.	59	93	158%
3.	82	89	109%
4.	121	194	160%
5.	79	96	121%
6.	795	957	120%
TOTAL	1179	1530	129%

Table 9: Experiment No.2 – Program written for a multiple user interactive system on a MOD 1 machine

The measurements given cover the "X" or "read only" segment of the program. The X segment comprises the program instructions together with constants and arrays having pre-set values.

Total Program Sizes:

<i>Assembler object size</i>	<i>CORAL 66 object size</i>	<i>CORAL 66 object as % of Assembler object</i>
1017	1286	126%

Table 10: Comparative Sizes of Main Modules and Internal Routines

<i>Program section</i>		<i>Assembler object size</i>	<i>CORAL 66 object size</i>
Main program		169	279
Routines	1	9	10
	2	140	155
	3	21	27
	4	68	64+
	5	163	215
	6	68	90
	7	14	12+
	8	95	115
	9	69	68
	10	44	46
	11	70	69+
	12	87	114

Note: + – reduction in size of the CORAL 66 version is due to the compiler organising a pool of constants relating to procedure names. In the Assembler version, these constants may appear in more than one routine.

APPENDIX 1

SOURCES OF DATA INCLUDED IN THIS REPORT

Admiralty Surface Weapons Establishment, Portsmouth, Hants.

Central Electricity Research Laboratories, Leatherhead, Surrey.

Computer Technology Ltd, Hemel Hempstead, Herts.

Defence Automatic Data Processing Training Centre, Blandford Camp, Dorset.

Ferranti Ltd, Bracknell, Berks.

National Physical Laboratory, Teddington, Middlesex.

Post Office, Goonhilly Radio Station, Helston, Cornwall.

Royal Aircraft Establishment, Farnborough, Hants.

Royal Armament Research and Development Establishment, Fort Halstead, Sevenoaks, Kent.

Royal Military College of Science, Shrivenham, Swindon, Wilts.

Royal Radar Establishment, Great Malvern, Worcs.

Signals Research and Development Establishment, Christchurch, Hants.

United Kingdom Atomic Energy Authority, AERE, Harwell, Berks.

APPENDIX 2

IMPLEMENTATION USING CORAL 66

A wide range of program types was found to have been implemented and to be under development using CORAL 66. Some of the areas for which CORAL 66 has been selected are listed below:

- Radar control
- Radio station monitoring
- Graphics support package
- Fire trajectory calculation
- Conversational real-time system
- Data reduction system
- Airborne computer systems
- Control of grid system
- Turbine and boiler control
- On-line heat-exchange monitoring

- Assembler for MOD 1
- CORAL 66 compilers
- FORTTRAN compiler

- Systems generation
- Executive interface package
- Executives

- Utility routines
- Text editing
- Disc utilities

- Information retrieval

See more recent list in Chapter 6 of main body of report.

ANNEX M

QUESTIONNAIRE

FOR A COMPARISON OF THE

PROGRAMMING LANGUAGES

CORAL 66
JOVIAL
LTR
PEARL

AND THE RESPECTIVE ANSWERS

The questions are a subset of the language comparison which was carried out by the LTPL-E committee, a subsection of the European branch of the "International Purdue Workshop on Industrial Computer Systems" (PURDUE EUROPE).

DEFINITIONS

Happenings

- connected happening — an occurrence generated as a direct effect of a PA
- free happening — an occurrence not generated as an effect of a PA but of interest to it
- loose happening — an occurrence generated as a side-effect of a PA, e.g. end-of-file, overflow. This corresponds to the PL/I "ON" condition.

Note: Messages which are generated by one PA for consumption by another are *connected* to the first, and *free* to the second. Only external interrupts are free to all PAs (and generally of interest to only one).

PAs

- parallel activity (PA) — the execution of a PCE
- synchronization of PAs — controlling PAs so as to maintain some desired sequential relationship between their effects
- independent PA — a PA (first) is independent from another PA (second) which has initiated the first, when the first might continue after the second has finished
- dependent PA — a PA (first) is dependent from another PA (second) which has initiated it, when the termination of the first is automatically connected with the termination of the second.

PCEs

- parallel code element (PCE) — segments which could be executed simultaneously by a multiprocessor.

Segment — a sequence of executable statements.

GENERAL

1. Give a brief history and bibliography!

CORAL 66

CORAL 66 was designed at the Royal Radar Establishment, Malvern, England in 1966 as a language which was suitable for use in process control and real time systems. It is partly derived from JOVIAL and ALGOL 60 and is intended to produce efficient object code. About 1970 it was adopted by the Ministry of Defence as a standard for weapon systems and "The Coral Group" was formed in 1973 to promote its use on a national basis within the United Kingdom, under the technical authority of the Royal Radar Establishment.

Reference:

P.M.Woodward, PR.Wetherall, B.Gorman
Official Definition of CORAL 66
Her Majesty's Stationary Office

JOVIAL**LTR**

1968-1969: Specifications of LTR.
1972: A prototype compiler and a prototype monitor are operational.
1973: Beginning of the implementation of a large real-time system.
1974: LTR is chosen as the standard language for operational programming in the French Armies.
1975: Multi-pass optimizing compilers are operational for:
IRIS 45, 50, 55, 60
IBM 360/370
Mitra 15

Bibliography:

LTR: manuel de references
MAXIRIS: real-time languages description manual
LTR: graphs syntaxiques.

PEARL

The language "PEARL" (process and experiment automation real time language) has been developed by a group of approx. 13 representatives of vendor companies, users, software houses and institutes in the FRG. This group was founded in 1969 and published a first concept¹ in 1970.

A detailed language description was published in April 1973 (Ref.2) and an introductory paper in September 1973 (Ref.3).

The work of the PEARL-group was sponsored by the German Ministry of Research and Development either directly or via the "Project PDV" (= Prozessdatenverarbeitung).

PEARL is a "middle level language" (like e.g. FORTRAN, ALGOL or PL/I) and intended to be used by e.g. the process engineer, physicist, chemist with little experience in programming.

Reference:

1. PEARL, The Concept of a Process and Experiment-oriented Programming Language; elektronische Datenverarbeitung, 10 (1970), 429-442.
2. PEARL, A proposal for a process- and experiment automation real time language; KFK-PDV1, April 1973.
3. PEARL, eine Prozess- und Experiment-orientierte Programmiersprache; Angewandte Informatik 9/73, 363-372.

* * *

2. For which class of processing problems was the language explicitly designed?
Find quotations from original reports.

CORAL 66

CORAL 66 was designed for use in process control and real time systems. It is particularly suited to areas of application where object code efficiency is of importance.

JOVIAL

"... JOVIAL is a language ... designed originally for the purpose of command and control system programming, its primary area of applicability is for the development and maintenance of large systems. JOVIAL is especially suited to systems requiring efficient processing of large volume of data of complex structure ..."

There is a mandatory set of target machine parameters for each implementation to be specified (e.g.: bits/word bytes/word address units/word etc.)
 [JOVIAL J73/1 Computer Programming Manual
 Computer Sciences Corporation]

LTR

LTR was explicitly designed for multi-tasks real-time applications.

PEARL

Chapter 0.3 of the PEARL-Report starts with the sentence: "Being a general-purpose realtime language, PEARL contains a quite complex set of task operations and mechanisms for timing and scheduling."

Further on one can read: "It is possible to describe completely even complex hardware configurations, to use symbolic names for hardware entities throughout the whole problem programs and to connect hardware interrupts to their software responses."

"PEARL contains possibilities of introducing user-defined data types and operators with restrictions for reasons of efficiency."

TASKING

1. *Is tasking included in the language?*

CORAL 66

Not language elements, the ability to use tasking features is given by inline code and macro features.

JOVIAL

There is no tasking in the language; the questions 2) – 11) are therefore not relevant.

LTR

There is a large set of tasking features.

PEARL

A large set of tasking features.

* * *

2. *Are Parallel Code Elements (PCEs) defined explicitly? If so, give example!*

CORAL 66

Not defined explicitly in CORAL but in the linkage mechanisms for system generation.

JOVIAL

Not relevant

LTR

Yes.

A PCE is defined as a process article:

```
ARTICLE PROCESS <name> (<parameters list>)
<block>
```

A process is reentrant.

Example:

```
ARTICLE PROCESS alpha (REAL VALUE x, y)
BEGIN ..... END
```

PEARL

No; PCEs are implicitly defined together with the declaration of PAs or are assigned to a PA at the time of activation.

Example:

```
MODULE;
PROBLEM;
TA1: TASK
TASKNAME TA2;
```

```

.
.
.
ACTIVATE TA2: BEGIN;
.
.
.
END;
.
.
.
END;
MODEND;

```

Remark:

The example also shows the possibility of "subtasking" in PEARL, i.e. the possibility of nested declaration of parallel activities. So the PA "TA2" is not known outside of "TA1". Operations on "TA1" affect the whole "tree" of subtasks as well.

* * *

3. *Is there a language difference between the declaration/execution of a PCE and the declaration/call of a procedure?
Please give examples for all four cases!*

CORAL 66

No difference (S.G.time).

JOVIAL

Not relevant

LTR

Yes.

.Declaration of a PCE: cf 2.

.Calling of an article process: it is the creation of a task on that process

Example:

alpha [a, b] OPEN PRIORITY 5;

.Declaration of a procedure:

ARTICLE PROCEDURE name (<parameters list>)

<block>

.Calling a procedure

<name> [<actual parameters list>];

PEARL

Declaration:

T1: TASK [attributes]

.

.

.

END;

P: PROCEDURE [attributes]

..... END;

Execution:

[schedule] ACTIVATE T1 [attributes];

CALL P [(parameters)];

Remark:

At the first glance there is no formal difference between the declaration of a PA and a Procedure. But a look at the attributes gives a first insight in the differences:

Attributes of a PA:

GLOBAL /* known outside module-level */

RESIDENT /* PCE will not be swapped out when PA not active */

PRIORITY [number]

Attributes of a Procedure:

parameter-list

return-attributes

M-6

GLOBAL /* c.f. above */
REENTRANT /* self explanatory */
RESIDENT /* code of Proc. will be loaded at load time of PCE which may call the Proc. */

There is also no equivalence on the procedure state to the declaration of a task-name.

The activation of a PA, compared to the call of a procedure reflects clearly the differences between the two mechanisms.

* * *

4. *What are the visible (and accessible) attributes of a Parallel Activity (PA)? (e.g. priority, state, residence)*

CORAL 66

Not in the language.

JOVIAL

Not relevant

LTR

A PA is defined as one execution of a process with given parameters and a given priority. (Several PA's can be active on the same process).

The attributes of a PA are:

- . its actual parameters
- . its priority.

PEARL

- task identifier
- priority
- residence

Remark:

C.f. question 3. The task identifier is no attribute in the strict sense, it gives the identity of the PA.

* * *

5. *Are PCE/PA operations included in the language? How?*

CORAL 66

Only by procedure calls.

JOVIAL

Not relevant

LTR

Yes, by statements.

PEARL

Yes, by statements.

* * *

6. *What commands can be executed by a PA on another PA (not itself)? Please give examples for each!*

CORAL

Not relevant.

JOVIAL

Not relevant.

LTR

A PA can

- . create a PA on another process
- . make a PA eligible (ready for execution)
by activating an event
or releasing a semaphore entry point.

PEARL

ACTIVATE – statements (Creation of another PA)

SUSPEND – statements

CONTINUE – statements

TERMINATE – statements (Destroy another PA)

PREVENT – statements (Destroy the schedule for another PA)

Example: WHEN interruptname ACTIVATE OTHERTASK;

* * *

7. What commands can be executed by a PA operating on itself? Please give examples for each!

CORAL 66

Not relevant.

JOVIAL

Not relevant.

LTR

A PA can

. create another PA on its own process

. put itself in a waiting state

Example:

WAIT 10;

WAITIO [CB]

. give back control to the scheduler:

MONITOR;

PEARL

SUSPEND

CONTINUE

RESUME

TERMINATE

PREVENT

Example: AFTER 5 MIN RESUME; /* This causes a delay of 5 minutes */

* * *

8. Is there a timing mechanism?

If yes, what can this mechanism schedule?

CORAL 66

No.

JOVIAL

Not relevant.

LTR

Yes.

That timing mechanism provides:

. the value of the real time clock: TIME

. delays that can be specified by an arithmetic expression.

Delays can schedule:

. eligibility of tasks (PA)

example:

alpha [a, b] OPEN PRIORITY 5 WAITING 10;

The PA will become eligible after 10 units of time.

. a task:

example: WAIT 10;

the task will be in a waiting state during 10 units of time.

. activation of events:

example:

ACTIVATE ev1 WAITING 5;

ev1 will occur after 5 units of time.

M-8

PEARL

Yes.

Execution of task-operations at specified points of time, when certain interrupts occur, after specified time intervals, cyclic execution (during certain time intervals), and combinations thereof.

* * *

9. *What types of happenings are distinguished in the language and how is each defined?*

CORAL 66

Not in the language.

JOVIAL

Not relevant.

LTR

The different types of happenings are:

- . end of a created task
- . end of an I/O operation
- . occurrence of an event
- . occurrence of an interrupt

PEARL

There are: free happenings, loose happenings, connected happenings.

Examples: loose happening ("SIGNAL"):
ON PRINTERERROR GOTO L;
free happening ("INTERRUPT"):
WHEN INTERRUPT (1)
ACTIVATE TA1;
connected happening ("SEMAPHORE"):
RELEASE SORTED;

* * *

10. *What facilities exist for explicit synchronization between PAs, or between PAs and I/O devices in the language?*

CORAL 66

Not relevant.

JOVIAL

Not relevant.

LTR

. Events

An event can be used as a parameter

Example

ARTICLE PROCESS p1

BEGIN REAL x1; EVENT ev1;

----- (1) -----

p2[x1, ev1] OPEN PRIORITY 2;

WAIT ev 1;

----- (2) -----

END

ARTICLE PROCESS p2 (REAL VALUE x; EVENT ev)

BEGIN

SETEV ev;

END

We have the following executions:

- . beginning of a task t1 on p1: (1)
- . the task t1 is put in a waiting state
- . execution of a task t2 on p2
- . activation of the event ev1;
- . end of the execution of t2: (2)

Semaphores.

Example: RESOURCE res1 USERS 1;
res2 USERS 4;

Semaphores can also be passed by parameters

Synchronisations between a PA and an I/O

Example: IOCS [CB];
WAITIO [CB];

The task will be put in a waiting state until the end of the I/O operation.

PEARL

Semaphores, Bolts and Interrupts.

Remarks:

Semaphores are intended as the usual means of explicit synchronisation between PA's in PEARL. There are the operations REQUEST ("P") and RELEASE ("V") on them. There is also the possibility of applying Semaphore-operations on lists of semaphores simultaneously, which help to avoid certain kinds of deadlocks.

The Bolts are an additional tool for explicit synchronisation. They allow discrimination between shared and exclusive access to resources on user level.

ENTER <u>boltname</u> ;	RESERVE <u>boltname</u> ;
LEAVE <u>boltname</u> ;	FREE <u>boltname</u> ;

There are also operations to induce signals and to trigger interrupts, but they are not intended for "normal" synchronisation purposes. They were included in PEARL in order to be able to write modules which can provide a test environment for other modules and for this purpose simulate the occurrence of events.

* * *

11. How can data be passed between PAs?

CORAL 66

Block structure and parameters.

GLOBAL

JOVIAL

Not relevant.

LTR

Data can be passed as parameters. Any type of data can be passed as a parameter, including events and semaphores.

PEARL

By means of variables that are in scope to the PA's and by files.

CONFIGURATION DESCRIPTION

1. How are I/O devices symbolically identified in the application program? Please give examples!

CORAL 66

Either as channel numbers or as strings, both being used as parameters, e.g. SETSINK ("LP");
SETSOURCE (1);

JOVIAL

There are no means for configuration description, the questions 1) - 10) are therefore not relevant.

LTR

I/O devices must be declared in the "ARTICLE SYSTEM DATA":

. DEVICE DIRECT <name> <address of the device>;
. DEVICE IOUNIT <name> <address of the device>;
. DEVICE INTRPT <name> <integer constant>;

PEARL

Not only devices, but also interrupts and signals are identified (in the same way as other program-elements) by identifiers chosen by the programmer and declared on one of the three reference levels (0: constant, 1: normal variable,

2: address of normal variable) with one of the attributes DEVICE, INTERRUPT, or SIGNAL respectively. (Also arrays are possible and all types mentioned are single or array — permitted as parts of structures). The connection data path between a declared identifier and the respective part of the system hardware is described by the "system-division" together with special related declarations within the "problem-division" of the PEARL-program.

Examples of a "system-division" and some device declarations within the "problem-division"

MODULE LIBRARY PUMPROCESS;

SYSTEM:

CP 1048 ↔ CH372;

CH372 *1*1, 16←ADC3;

*3*1, 8←DID5;

*4*1, 8→DOD2;

*5*1, 8→AUSGABE:FS3;

MANOMETER: →ADC3 *57;

INDIKATOR: →DID5 *72*5,1;

VENTSTATUS: → *72*6,2;

TASTE → *78*1,2;

PUMPE: ←DOD2 *53*3,2;

VENTIL: ← *53*5,2;

PRINTER: SIG(10) ←;

ALARM: ITR (5) ←;

PROBLEM;

DCL (TASTE, PUMPE, VENTIL) VAL DEVICE GLOBAL;

DCL PRINTER VAL SIGNAL GLOBAL, ALARM VAL INTERRUPT GLOBAL;

.
.

.

* * *

2. *Is it possible to connect these symbolic identifiers with external designations, e.g. vendor's designations?*
 - *Is it necessary?*
 - *When is it done?*
 - *By what mechanism? Please give examples!*
 - *How is it possible to modify such a connection at run-time?*

CORAL 66

As the example in I shows this can be done if the system has been so implemented. As there are so many implementations it is not possible to answer the subquestions individually.

JOVIAL

Not relevant.

LTR

It is possible to connect at compile-time only.

PEARL

— It is necessary to connect the symbolic identifiers with implementation defined designations. In the "system-division" the programmer describes the parts of the system the program will use at run-time. These parts to which he wants to refer explicitly (in the "problem-division") have to be named by him.

These names can be used within the "problem-division" for several purposes. It is possible to define device arrays (e.g. for process applications).

— The mentioned "system-division" has to be given at "programming time". It is part of the source-program.

— Incomplete sample program:

MODULE EXAMPLE;

SYSTEM;

CPU↔CHANNEL: implementation defined designation

CHANNEL * 3 → PRINTER: LP # 701;

(PRINTER=name chosen by the programmer)

.
.

.

PROBLEM;

.


```

DECLARE PRINTER VAL DEVICE GLOBAL;
.
DECLARE VARIABLE DEVICE; (Reference 1)
DECLARE CONST VAL DEVICE = IDENTICAL PRINTER;
.
.
.
VARIABLE = CONST;
.
.
.
PUT TO PRINTER . . .;
PUT TO CONST . . .;
PUT TO VARIABLE . . .;

```

- The specifications given by the "system-division" cannot be changed at run-time (evidently).

Device switching can be done e.g. by using assignments to reference - 1 - identifiers declared with the respective device-attribute or by using a variable index within a reference to a device array.

* * *

3. *Explain how the routes (data paths) from the computer to the external devices are identified!*

CORAL 66

Not possible on language level.

JOVIAL

Not relevant.

LTR

The programmer specifies only the addresses of the devices for the data channels.

PEARL

See answer to question 1.

* * *

4. *Is it possible to specify hardware information about I/O devices?*
 - *Is it necessary?*
 - *What is this information?*
 - *When is it given?*
 - *By what mechanism? Please give examples!*

CORAL 66

No.

JOVIAL

Not relevant.

LTR

?

PEARL

The full hardware information of any device is implicitly contained within the implementation defined designations of the system components used for constructing the "system-division" of the PEARL-program.
For each implementation these designations and information have to be presented to the programmer in a system manual.
Therefore:

- No
- All hardware information concerning the respective system component.
- At programming time.
- See the example given in the answer 2.

* * *

5. *Are hardware- or system happenings provided in the language?*

CORAL 66

In some implementations the use of ON commands is allowed,
e.g.: ON INTERRUPT (7) DO procedure.

JOVIAL

Not relevant.

LTR

Yes.

LTR provides interrupts and events.

PEARL

There exist interrupts and signals in PEARL. They differ in the way of programming reactions to them and in the respective attributes within the declaration.

Remark:

The interrupts are "free happenings" and the signals "loose happenings" in the sense of the definitions (see page 1).

* * *

6. *How are hardware- or system happenings symbolically identified in the language?*

CORAL 66

By the ON command method when allowed.

JOVIAL

Not relevant.

LTR

?

PEARL

See answer to question 1.

* * *

7. *How are these symbolic identifiers connected with hardware devices?*

— *When is such a connection made?*

— *How can this connection be modified at run-time?*

Please give examples!

CORAL 66

Via numbers identifying channels.

— At program generation or start time.

— No, cannot be modified at run-time.

JOVIAL

Not relevant.

LTR

A symbolic name is associated with an interrupt level by the means of the DEVICE INTRPT declaration

PEARL

See answer to question 2.

Remarks:

— At programming time

— The connection can be made in the same way as it is done in the case of devices, i.e. by assignment to "interrupt variables".

* * *

8. *Is it possible to describe the application in order to build a specific O.S. for a specific application?*

If yes, then:

- By what mechanism?*
- What are the main elements of such a description?*
- When and how are they specified?*
- Is it possible to optimize O.S. modules, e.g. by additional assembling?*

If no, then:

- Show how a general O.S. is optimized according to a specific application!*

CORAL 66

Not in the language, purely an operating system feature.

JOVIAL

Not relevant.

LTR

LTR systems usually run under a LTR multi-tasks monitor. These monitors are modular and a monitor generation can be made for the application.

PEARL

The "system-division" in PEARL is a means for tailoring a specific (and optimized) operating system for an application out of a modular "master - O.S."

- In the "system-division" the programmer defines the required configuration by means of device descriptions and the description of the hardware connection.

Example:

CPU ↔ CHAN;

CHAN * 5 → TERMINAL : TTY (1);

(TERMINAL is the symbolic name for Teletype 1, which is connected to channel 5 of the CPU).

- The elements of such a description are
 1. Implementation defined "device types" (CPU, CHAN, TTY).
 2. User defined symbolic names (TERMINAL)
 3. CONNECTION symbols (*5, →)
- The configuration description is made at programming time.
- Outside the scope of the language, therefore no.
- The output of the compilation of the system description may deliver a set of parameters which allow to include (resp. exclude) the needed (resp. not needed) O.S. modules at system-generation time.

* * *

9. *How is it possible to describe the necessary resources for a PA to run?*

- When is it done?*
- Can it be changed at run-time?*

CORAL 66

Either by routes and facilities described at system generation time or at program start time. Some resources such as core may also be allocated at run-time. The time at which resources are made available and changed depends on the type of resource. For example, a system might allow core to be allocated as necessary at run-time but only up to a limit set at system generation time. A message path to another PA may only be set up at system generation or PA initiation time.

JOVIAL

Not relevant.

LTR

It is only possible to control competitions for the resources by the means of semaphores.

PEARL

Core memory and processing time management has mainly to be done automatically by the O.S. but the programmer has the possibility to attach the attribute RESIDENT to a PA when it is declared.

For competition between PAs priorities can be assigned at the declaration at the activation (start) or at the continuation of the respective PAs. Management of other resources can be done by using bolts and the operations RESERVE, FREE, ENTER and LEAVE which will be executed at run-time, or by using semaphores.

* * *

10. *Is it possible to use existing features in the language for I/O devices which need special hardware interfaces not provided by the vendor of the application computer?*

CORAL 66

Yes, as all I/O is done via such existing features.

JOVIAL

Not relevant.

LTR

It is usually possible using the IOCS statement with special control block and writing a special driver.

PEARL

Possible by "TAKE/SEND-statement". Transfer of command pattern as data until the point where O.S. support ends.

Remark:

The philosophy behind this technique is, that any device, it may be as alien as it wants, has to be connected to some standard interface (e.g. channel) of the computer, which has O.S. support.

INPUT/OUTPUT

1. *Please describe in short form the scope of the set of I/O facilities in the language!*
 (As guidelines:
"only simple mechanisms: I/O only of single data items, only binary transfer, only single hardware registers addressable" etc. or
"a comprehensive system with character-, graphic-, process-I/O, different formats" etc. . . .)

CORAL 66

The I/O facilities being based on the Code Statements normally called up by Macros or system procedures are capable of making the optimum use of any hardware facility on any particular machine. Hence, by a properly constructed set of system Macros or Procedures, the I/O facilities can be made to be as comprehensive as desired.

JOVIAL

There are no I/O-facilities on language level, the questions 2) – 10) are therefore not relevant.

LTR

There are 4 levels of I/O facilities:

1. Direct input-output (of one data word)
 Example: `DEVICE DIRECT d1 00F0;`
`INPUT [d1, Info, logname];`
2. Supervisor controlled input-output
`IOCS [CB, logname];`
 CB is the name of a "control block". It may be the name of a structure or a reference to a set element in dynamical storage.
3. Formatted input-output
`READ {`
`WRITE {` name1 name2 [<variables list>];
 name1: name of a logical file
 name2: name of a FORMAT declaration.
4. Specific input-output

PEARL

The data sources and data sinks which communicate by the flow of data in a process control system may be physical devices (e.g. disk memory, ADC) or logical devices (e.g. files, working storage).

The programmer can address a physical device if its name occurs in the system-division. Logical devices require additional organization. In the case of files storage areas are established by file-handling-statements. In the case of working storage the compiler is responsible for this organization.

PEARL provides a complete set of conventional I/O together with tools for graphic data handling and inter-system data transfer.

2. *What are the differences between process I/O and conventional I/O in the language?*

CORAL 66

None.

JOVIAL

Not relevant.

LTR

INPUT, IOCS, specific I/O can be considered as process I/O. READ and WRITE can be considered as conventional.

PEARL

Not-formatted-not-organized communication is the simplest form of communication in PEARL. The TAKE/SEND/MOVE-statements can transport data in binary form without any transformation.

* * *

3. *In which way is graphic I/O handled within or above the language?*

CORAL 66

Not in the language but handled via packages.

JOVIAL

Not relevant.

LTR

Graphic I/O can only be handled by the means of specific global procedures.

PEARL

There are special statements for graphic I/O in the language:

DRAW TO graph-device FROM data-source BY format;

SEE FROM graph-device TO data-sink BY format;

* * *

4. *Can all external device registers be addressed and accessed via I/O operations or some other control operations?*

CORAL 66

Depends on system implemented.

JOVIAL

Not relevant.

LTR

External device registers are not usually under program control.

PEARL

All external devices, which are described in the "system-division" can be addressed by the program. Special functions (like "test", "initialize") can be performed by transferring a respective bit-pattern to the "outermost" point where those can still be regarded as normal I/O-commands (c.f. Question 10 in the configuration description chapter), or by using options within a "data-less" TAKE/SEND statement.

* * *

5. *Which possibilities are there to switch between devices or data paths at run-time?*

CORAL 66

Depends on operating system but useful for debugging and allowed in RRE systems.

JOVIAL

Not relevant.

LTR

Only for supervisor controlled input-output by changing the CB (control block).

PEARL

The connection between a device-identifier and the device is made at compile time and can't be changed. But the actual device for an I/O operation is defined by the value of the device-identifier which may be a reference and thus can be changed (c.f. also question 2 of the configuration description chapter).

* * *

6. *How is I/O of high data rates managed?*

CORAL 66

Normally by using an array and handing the address of the first item and the length to the operating system.

JOVIAL

Not relevant.

LTR

By the means of the IOCS statement of the specific I/O statements.

PEARL

Any external device or channel can be described in the "system-division" and can therefore be used within the "problem-division". Since internal sources or sinks may be arrays, high data rates can be achieved if the hardware is capable of handling them.

* * *

7. *Which are the formatting facilities concerning process data?*

CORAL 66

Depends on system implemented.

JOVIAL

Not relevant.

LTR

The formatting facilities allow binary-alphanumeric conversions for all the data types.

PEARL

Modification of transferred data is only possible by options within the respective I/O-statement.

Remark:

This option also allows to include special hardware features (like e.g. "read-and-clear") on language level, although in an implementation dependent manner.

* * *

8. *How is status information about I/O operations and/or devices accessible by the program? Can (or must) it be taken into account before initiation of a new I/O operation?*

CORAL 66

Status information can be returned as the answer to a library procedure.

JOVIAL

Not relevant.

LTR

For an I/O operation initialized by the means of an IOCS statement, the task can wait the end by means of the **WAITIO** statement.

An interrupt procedure executed when the end of I/O interrupt occurs can activate a software event.

A new I/O can be initiated without waiting for the end of the previous one.

PEARL

Response may occur during the execution of an I/O statement. They are not (yet) standardized.

Remark:

They can be accessed and responded to by the "signal-response-statement":

ON signal-name statement;

This statement has to precede the I/O-statement during execution of which the respective signal is expected.

* * *

9. *How are process interrupts handled?*

CORAL 66

Depends on operating system.

JOVIAL

Not relevant.

LTR

An interrupt level must be declared in the "article system data":

DEVICE ENTRPT <name> <level>;

It is possible to control the status of the interrupt flip flop using the

ITCTRL statement.

It is possible to connect an interrupt procedure to an interrupt level using the

ATTACH statement

and to make the disconnection using the

DETACH statement.

PEARL

Like all other possible interrupts, they may be connected with a WHEN-schedule to a task.

Example:

WHEN ALARM ACTIVATE T1;

* * *

10. *How are sources of process interrupts identified?*

CORAL 66

Depends on operating system but normally as replies to system calls.

JOVIAL

Not relevant.

LTR

?

PEARL

The connection between an interrupt and a device may be described in the "system-division" of the PEARL-program.

ALGORITHMIC FEATURES

1. *What kinds of data types can be handled and how are they described?*

CORAL 66

FIXED (n, m) n = width, m = scale (precision)

FLOATING

INTEGER

TABLES as described at question 3 are also allowed

ARRAYS OF FIXED, FLOATING or INTEGER items.

JOVIAL

The following data types exist for declarations:

- Character strings (length in bytes)
- Floating Point (length in bits)
- Signed Integers (length in bits)
- Unsigned Integers (length in bits)

There are constant denotations for bit strings for use in bit expressions.
Unsigned integers can be used also in bit expressions.

LTR

Data type	Declaration
Integer	INTEGER
Fixed point number	FIXED <scale>
Floating point number	REAL
Bits string	LOGICAL
Boolean	BOOLEAN
Character string	STRING
Pointers	REFERENCE
Quality	QUALITY

Quality: a type of data that can take a finite number of attributes:

Example:

QUALITY (RED 1, GREEN 2, BLUE 3) COLOR;

PEARL

In PEARL one can handle fixed point and floating point data with unlimited precision. For a specific variable the programmer has to define a specific precision (or use the implementation defined precision by default). The key-words for arithmetic data are FIXED and FLOAT.

In PEARL one can handle bit-patterns with arbitrary length. For each variable of mode BIT one has to choose a specific number bitpositions or use the implementation defined length by default. It is also possible to handle character strings of arbitrary length. Variables with the attribute CHARACTER are of programmer defined length or implementation defined length. Variables may be connected also with attributes like CLOCK, DURATION, DEVICE, INTERRUPT, SIGNAL.

* * *

2. *Is it necessary to declare the affiliation connected to an identifier explicitly or are there any implicit declarations?*

CORAL 66

No implicit declarations except for labels.

JOVIAL

Affiliations: data allocation
 " type
 " size
 array structure
 packing mode
 .
 .
 .
 etc

There is no implicit declaration except for labels, but some of the attributes are optional and filled in by default.

LTR

Declaration is mandatory.

PEARL

No implicit declaration except for labels.

* * *

3. *Is it possible to handle as one item data collections or structures, consisting of components of different types?*

CORAL 66

Yes, "TABLE". A declaration of a "TABLE" may be of the following nature.

```
"TABLE" PERSON [3, 12]
    {SEX "UNSIGNED" (1) 0,8;
    AGE "UNSIGNED" (8) 0,0;
    HEIGHT "INTEGER" 1;
    WEIGHT "FIXED" (16,4) 2
"PRESET" (1, 23, 72, 156), (0, 21, 63, 112), (0, 28, 65, 126), (etc.)
    (      ), (      ), (      ), (      )
    (      ), (      ), (      ), (      )];
```

This gives twelve 3 word records. The first word is broken into two fields which can be referenced by name. The next two words are used whole. Note the method of initializing the structure (optional).

JOVIAL

Structures may be passed as parameters; there is no other referencing possibility. Structures are introduced by means of data-BLOCKS, e.g.:

```
BLOCK BLOCKNAME; BEGIN ITEM NAME1 C2
                        ITEM NAME2 S10
                        ITEM NAME3 U11
                        END
```

LTR

Yes. There are two possibilities:

- . A structure is a collection of components of different types:
STRUCT s (REAL x,y; SHORT INTEGER t);
- . A set is a list of structured items in dynamical storage:
SET t (REAL a,b; SHORT INTEGER c);

PEARL

Yes. The description is introduced within the declaration. As an example: DCL RS STRUCT (VALUE FLOAT, NAME CHAR (12)); specifies a structure (a variable of type structure). RS refers to two elements, one whose mode is real and another whose mode is string and which is capable of receiving up to twelve characters.

* * *

4. *How are new data types defined?*

CORAL 66

Not possible to define new data types in ALGOL 68 style using mode.

JOVIAL

There is no possibility to define new data types.

LTR

It is not possible to define new data types.

PEARL

By introducing a structure as a new data type.

Example:

```
TYPE COMPLEX = STRUCT (RE FLOAT, IM FLOAT);
```

* * *

5. *How are address variables declared?*

CORAL 66

They are declared as INTEGERS used as address variables by the LOCATION facility which returns the address of an item and the anonymous reference which allows the contents of an INTEGER variable to be used as an address.

Example: i: = "LOCATION" (var);

```
[i]: = 1;
```

"COMMENT" would set contents of var to 1;

JOVIAL

Integers can be used as pointers. The location function (see answer 7) delivers physical addresses of program entities which then can be assigned to integers.

LTR

As references:

Examples:

```
REFERENCE x ref1;
ref1 contains the address of x.
REFERENCE ANY ref2;
```

PEARL

They are declared in the same manner as normal variables are introduced. The keyword REF must precede the type attribute.

Example: DCL AA REF FLOAT;

* * *

6. *By what mechanism are the elements of a structure selected?*

CORAL 66

A "TABLE" is a vector of fixed length records of the same format. Access to any element is made by quoting the name of the item required followed by the index to the record containing the desired item.

Example: using "TABLE" declaration in question 3 and assuming I, CHARLES are some variables, reference is made to table as follows: I := AGE [CHARLES];

JOVIAL

The elements of data-BLOCKS have their own names by which they can be referenced, this is also possible by using the BLOCK-name as base address.

LTR

. The elements of a STRUCTURE are selected by their names:

Example (c.f.3)

```
x = 0; y = 0;
```

. The fields of an element of a set are selected through a reference:

Example (c.f.3)

Creation of an element: NEW ref2 in t;

```
a (ref3) = 0; b (ref3) = 0;
```

PEARL

In the example given in 3 the float number is selected by RS.VALUE and the character string by RS.NAME.

* * *

7. *Which operators and built-in functions exist for the different data types and aggregates?*

CORAL 66

```
+ - * / "DIFFER" "UNION" "MASK"
"BITS" "LOCATION" = "AND" "OR"
<= >= <> = < >
```

JOVIAL

Intrinsic functions:

- Bit extraction out of data-values
- Byte extraction out of data values
- Address investigation for any program entity
- Absolute value
- Sign investigation
- Bit shift
- Data size investigation
- Procedure data area size investigation

Operators

- Addition
- Subtraction

- Multiplication
- Division
- Exponentiation
- Modulo
- Comparison equal
- Comparison unequal
- Comparison less than
- Comparison greater than
- Comparison greater than or equal
- Comparison less than or equal
- NOT, AND, OR, XOR, EQV
- At-address

LTR

OPERATORS				
Arithmetic	Logical	Boolean	Reference	Comparison
EXP	CPL	NOT	AFTER	EQL
*	SLL	AND	BEFORE	NEQ
/	SRL	OR		GTR
DIV	SLA			GEQ
MOD	SRA			LSS
+	SLC			LEQ
-	SRC			

SLL, . . . , SRC are shifting operators on bits strings.

CPL: logical complementation.

PEARL

Besides the common arithmetic and logical operators there exist several operators for bit- and character-handling (such as selection, concatenation, shift etc.). There are also the usual built-in functions like SINE, COSINE, etc.

* * *

8. *How are new operators defined?*

CORAL 66

Must use procedures to implement new types of operation such as array addition.

JOVIAL

No possibility to define new operators.

LTR

There exists an operator declaration. By such a declaration either an existing operator token may be used to be connected to another routine so that this operator may be used with variables of a mode which until then would have been rejected by the compiler (e.g. the adding operator and variables of mode array), or completely new tokens or indicants (like identifiers for variables) may be introduced and connected with some special routine.

The following example shows how an operator declaration works:

Within the range of the two declarations

```
TYPE COMPL = STRUCT ( ( RE, IM) FLOAT);
```

```
OPERATOR + (A, B) RETURNS COMPL
```

```
  DCL (A, B) COMPL;
```

```
  RETURN ( (A.RE + B.RE), (A.IM + B.IM) );
```

```
END;
```

The application is possible:

```
DCL (Y, Z) COMPL INITIAL ((1, 2), (3, 4));
```

```
Z = Z + Y;
```

where after the elaboration Z refers to (4, 6) or 4 + i6 as result of the complex addition.

* * *

9. *How are procedures declared?*

CORAL 66

```
"TYPE" "PROCEDURE" NAME (Formal parameter list);
```

```
Procedure Body;
```

The "TYPE" is optional and is "INTEGER"
 "FIXED" (n, m)
 or "FLOATING"

The Procedure body is a single statement which may, however, be a Block containing its own local declarations.

JOVIAL

e.g.:

```
PROC EXAMPLE (PAR1, PAR2, RETURNPAR)
BEGIN
```

```
  .
  .
  .
  .
```

data and code

```
  .
  .
  .
```

```
END
```

LTR

There are procedures and typed procedures:

```
. ARTICLE PROCEDURE <name> (<parameters list>)
  <block>
. ARTICLE <type> PROCEDURE
  <name> (<parameters list>)
  <block>
```

PEARL

Example for a subroutine:

```
P1: PROCEDURE (A FLOAT, B DURATION)
```

```
  .
  .
  .
  .
```

```
END;
```

Example for a function:

```
P2: PROCEDURE (C FIXED) RETURNS (FLOAT)
```

```
  .
  .
  .
  .
```

```
END;
```

* * *

10. *What kinds of loops are programmable?*

CORAL 66

"FOR" VARIABLE :=FORLIST "DO" statement;

Note 1, the variable is exhaustively assigned each element of the arbitrary length FORLIST in turn.

Note 2, each element may be either

- 1) expression
- 2) expression "WHILE" condition
- 3) expression "STEP" expression "UNTIL" expression where 2, 3 may cause several loops for the same FOR element.

Note 3, the statement may be compound

JOVIAL

There are 2 kinds of loops: a while loop and a for loop e.g.:

```
WHILE x = y BEGIN ... END
FOR I: 1 BY 3 WHILE I <= 17 BEGIN ... END
FOR I: 17 THEN I + 4 WHILE I < 255 BEGIN ... END
```

The default of the counter initialization is the current value of the counter at entrance into the loop.

LTR

```

· WHILE <boolean expression> DO <statement>;
· FOR <name exp> = <arithmetic exp>
  WHILE <boolean exp> DO <statement>;
· FOR <name exp> = <arithm. exp>
  STEP <arithm. exp> UNTIL <arithm. exp>
  DO <statement>;

```

Example:

```

FOR x = x+dx WHILE f[x] GTR 0
DO s = s+g[x];

```

PEARL

There is a universal loop statement defined:

FOR identifier

```

FROM integer expression
BY integer expression
TO integer expression
WHILE binal (1) expression

```

REPEAT statement . . . END;

The optional parts here are to be understood as follows:

From the complete form

FOR I FROM A BY B TO C WHILE D DO E DONE;

The term "FOR I" may be omitted, if I does not occur in D OR E;

If A is a constant with the value 1, "FROM A" may be omitted and "FROM 1" is assumed by default. If B is a constant with the value 1, "BY B" may be omitted and "BY 1" is assumed by default. If D is a constant with the value B "1" (true), "WHILE D" may be omitted and WHILE B "1" is assumed. If no upper bound is required "TO C" may be omitted. This enables the implementer to produce optimal code for each kind of loop under the control of the programmer.

* * *

11. *What are the conditional statements, what do they look like and how can they be replaced by other language features?*

CORAL 66

Conditional statement is of form:

"IF" condition "THEN" any statement except a for statement or a conditional statement "ELSE" statement;

Note that the statements following "THEN", "ELSE" may each be a block (see 84).

JOVIAL

The form of the only conditional statement is:

IF conditional-expression; statement ELSE statement

(where "statement" may be BEGIN . . . END)

The ELSE-branch is optional.

LTR

```

· IF <boolean exp> THEN <statement>;
· IF <boolean exp> THEN <statement 1>
  ELSE <statement 2>;
· CASE <arithm. exp> OF
  BEGIN <statement 1>;
    <statement 2>;

```

END;

Note:

A Block is a statement.

PEARL

The conditional statement is defined:

IF binal (1)-expression THEN statement . . . ELSE statement . . . FIN; its elaboration is the same as in other languages.

The other conditional statement is the case statement:

CASE integer expression IN statement, statement, OUT statement . . . FIN;

The case statement serves to shorten nested conditional statements. In its simple form it serves as a computed goto, e.g. CASE I IN L1, L2, L3, L4 FIN;
that is a FORTRAN statement: GOTO (L1, L2, L3, L4), I

* * *

12. *How is in-line assembly language code handled?*

CORAL 66

The reserved word "CODE" introduces in-line code which is written as a "BEGIN" - "END" block immediately following.

This code is often put in a macro definition with the macro name giving the code some significance.

JOVIAL

No statement about in-line assembly code was found within the language description.

LTR

CODE

assembly code

;

PEARL

There is nothing defined in this direction in the report.

Remark:

In-line assembly code was regarded as a very dangerous feature by most of the authors of PEARL and therefore it was decided that nearly every capability one could reasonably expect from in-line code should be provided for by higher-level-language elements. But it is not forbidden for a courageous implementer to add some code-inclusion feature.

* * *

13. *What kinds of facilities are available for setting strings of source code (MACRO prototypes) to be expanded in standard form at compile time according to a model (MACRO model)?*

CORAL 66

CORAL 66 specifies a macro facility which allows parameters, and nesting of macro definitions to any depth, and further specifies deletion and re-definition should be possible.

JOVIAL

There is a "... source macro capability which allows for creating and modifying source by substitution of a parameterized source string ..."

LTR

.MACDEF <name> % characters string %;

Example:

MACDEF field % 1(2, 1)%;

.MACPRO: macro definition with parameters.

PEARL

There are no other than the TYPE and OPERATOR declarations, which are not MACRO facilities in the sense that a precompilation is done to change them into legal language statements, but could be regarded as such with respect to their power.

REPORT DOCUMENTATION PAGE			
1. Recipient's Reference	2. Originator's Reference AGARD-AR-90	3. Further Reference ISBN 92-835-1244-8	4. Security Classification of Document UNCLASSIFIED
5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 rue Ancelle, 92200 Neuilly sur Seine, France		
6. Title	A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS		
7. Presented at			
8. Author(s) Various			9. Date May 1977
10. Author's Address Various			11. Pages 560
12. Distribution Statement	This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Covers of all AGARD publications.		
13. Keywords/Descriptors	<div style="display: flex; justify-content: space-between;"> <div> Flight control Digital systems Guidance computers </div> <div> Standardization Data transmission Programming languages </div> <div> Standards Computer systems programs </div> </div>		
14. Abstract	<p><i>an investigation of the</i></p> <p>This Report contains the findings of the AGARD GCP Working Group No.02, set up to investigate standardization methods for digital guidance and control systems, particularly with regard to data transmission techniques and high level programming languages. It includes discussion of the general problems and techniques, reports on the particular experiences of the individual nations, and concludes that, whilst much work remains to be done on software aspects, the field of data transmission may be amenable to early standardization.</p> <p>The Annexes to the Report contain full details of the techniques studied, and include comparisons of data transmission methods and high level languages. These comparisons are not intended as quantitative assessments but are designed to outline the relevant features of the different techniques.</p>		

<p>AGARD Advisory Report No.90 Advisory Group for Aerospace Research and Development, NATO</p> <p>A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS Published May 1977 560 pages</p> <p>This Report contains the findings of the AGARD GCP Working Group No.02, set up to investigate standardization methods for digital guidance and control systems, particularly with regard to data transmission techniques and high level programming languages. It includes discussion of the general problems and techniques, reports on the particular experiences of the individual nations, and concludes that, whilst much work remains</p> <p>P.T.O.</p>	<p>AGARD-AR-90</p> <p>Flight control Digital systems Guidance computers Standardization Data transmission Programming languages Standards Computer systems programs</p>	<p>AGARD Advisory Report No.90 Advisory Group for Aerospace Research and Development, NATO</p> <p>A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS Published May 1977 560 pages</p> <p>This Report contains the findings of the AGARD GCP Working Group No.02, set up to investigate standardization methods for digital guidance and control systems, particularly with regard to data transmission techniques and high level programming languages. It includes discussion of the general problems and techniques, reports on the particular experiences of the individual nations, and concludes that, whilst much work remains</p> <p>P.T.O.</p>	<p>AGARD-AR-90</p> <p>Flight control Digital systems Guidance computers Standardization Data transmission Programming languages Standards Computer systems programs</p>
<p>AGARD Advisory Report No.90 Advisory Group for Aerospace Research and Development, NATO</p> <p>A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS Published May 1977 560 pages</p> <p>This Report contains the findings of the AGARD GCP Working Group No.02, set up to investigate standardization methods for digital guidance and control systems, particularly with regard to data transmission techniques and high level programming languages. It includes discussion of the general problems and techniques, reports on the particular experiences of the individual nations, and concludes that, whilst much work remains</p> <p>P.T.O.</p>	<p>AGARD-AR-90</p> <p>Flight control Digital systems Guidance computers Standardization Data transmission Programming languages Standards Computer systems programs</p>	<p>AGARD Advisory Report No.90 Advisory Group for Aerospace Research and Development, NATO</p> <p>A STUDY OF STANDARDIZATION METHODS FOR DIGITAL GUIDANCE AND CONTROL SYSTEMS Published May 1977 560 pages</p> <p>This Report contains the findings of the AGARD GCP Working Group No.02, set up to investigate standardization methods for digital guidance and control systems, particularly with regard to data transmission techniques and high level programming languages. It includes discussion of the general problems and techniques, reports on the particular experiences of the individual nations, and concludes that, whilst much work remains</p> <p>P.T.O.</p>	<p>AGARD-AR-90</p> <p>Flight control Digital systems Guidance computers Standardization Data transmission Programming languages Standards Computer systems programs</p>

<p>to be done on software aspects, the field of data transmission may be amenable to early standardization.</p> <p>The Annexes to the Report contain full details of the techniques studied, and include comparisons of data transmission methods and high level languages. These comparisons are not intended as quantitative assessments but are designed to outline the relevant features of the different techniques.</p>	<p>to be done on software aspects, the field of data transmission may be amenable to early standardization.</p> <p>The Annexes to the Report contain full details of the techniques studied, and include comparisons of data transmission methods and high level languages. These comparisons are not intended as quantitative assessments but are designed to outline the relevant features of the different techniques.</p>
<p>ISBN 92-835-1244-8</p> <p>to be done on software aspects, the field of data transmission may be amenable to early standardization.</p> <p>The Annexes to the Report contain full details of the techniques studied, and include comparisons of data transmission methods and high level languages. These comparisons are not intended as quantitative assessments but are designed to outline the relevant features of the different techniques.</p>	<p>ISBN 92-835-1244-8</p> <p>to be done on software aspects, the field of data transmission may be amenable to early standardization.</p> <p>The Annexes to the Report contain full details of the techniques studied, and include comparisons of data transmission methods and high level languages. These comparisons are not intended as quantitative assessments but are designed to outline the relevant features of the different techniques.</p>
<p>ISBN 92-835-1244-8</p>	<p>ISBN 92-835-1244-8</p>

AGARD

NATO  OTAN

7 RUE ANCELLE · 92200 NEUILLY-SUR-SEINE
FRANCE

Telephone 745.08.10 · Telex 610176

**DISTRIBUTION OF UNCLASSIFIED
AGARD PUBLICATIONS**

AGARD does NOT hold stocks of AGARD publications at the above address for general distribution. Initial distribution of AGARD publications is made to AGARD Member Nations through the following National Distribution Centres. Further copies are sometimes available from these Centres, but if not may be purchased in Microfiche or Photocopy form from the Purchase Agencies listed below.

NATIONAL DISTRIBUTION CENTRES

BELGIUM

Coordonnateur AGARD - VSL
Etat-Major de la Force Aérienne
Caserne Prince Baudouin
Place Dailly, 1030 Bruxelles

CANADA

Defence Scientific Information Service
Department of National Defence
Ottawa, Ontario K1A 0Z2

DENMARK

Danish Defence Research Board
Østerbrogades Kaserne
Copenhagen Ø

FRANCE

O.N.E.R.A. (Direction)
29 Avenue de la Division Leclerc
92 Châtillon sous Bagneux

GERMANY

Zentralstelle für Luft- und Raumfahrt-
dokumentation und -information
Postfach 860880
D-8 München 86

GREECE

Hellenic Armed Forces Command
D Branch, Athens

ICELAND

Director of Aviation
c/o Flugrad
Reykjavik

ITALY

Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
3, Piazzale Adenauer
Roma/EUR

LUXEMBOURG

See Belgium

NETHERLANDS

Netherlands Delegation to AGARD
National Aerospace Laboratory, NLR
P.O. Box 126
Delft

NORWAY

Norwegian Defence Research Establishment
Main Library
P.O. Box 25
N-2007 Kjeller

PORTUGAL

Direcção do Serviço de Material
da Força Aérea
Rua de Escola Politécnica 42
Lisboa
Attn: AGARD National Delegate

TURKEY

Department of Research and Development (ARGE)
Ministry of National Defence, Ankara

UNITED KINGDOM

Defence Research Information Centre
Station Square House
St. Mary Cray
Orpington, Kent BR5 3RE

UNITED STATES

National Aeronautics and Space Administration (NASA),
Langley Field, Virginia 23365
Attn: Report Distribution and Storage Unit

THE UNITED STATES NATIONAL DISTRIBUTION CENTRE (NASA) DOES NOT HOLD
STOCKS OF AGARD PUBLICATIONS, AND APPLICATIONS FOR COPIES SHOULD BE MADE
DIRECT TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS) AT THE ADDRESS BELOW.

PURCHASE AGENCIES

Microfiche or Photocopy

National Technical
Information Service (NTIS)
5285 Port Royal Road
Springfield
Virginia 22151, USA

Microfiche

Space Documentation Service
European Space Agency
10, rue Mario Nikis
75015 Paris, France

Microfiche

Technology Reports
Centre (DTI)
Station Square House
St. Mary Cray
Orpington, Kent BR5 3RF
England

Requests for microfiche or photocopies of AGARD documents should include the AGARD serial number, title, author or editor, and publication date. Requests to NTIS should include the NASA accession report number. Full bibliographical references and abstracts of AGARD publications are given in the following journals:

Scientific and Technical Aerospace Reports (STAR),
published by NASA Scientific and Technical
Information Facility
Post Office Box 8757
Baltimore/Washington International Airport
Maryland 21240, USA

Government Reports Announcements (GRA),
published by the National Technical
Information Services, Springfield
Virginia 22151, USA



Printed by Technical Editing and Reproduction Ltd
Harford House, 7-9 Charlotte St, London W1P 1HD

ISBN 92-835-1244-8

AR 90

TECHNICAL METHODS